Technische
Universität
Berlin

Fakultät für Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Lehrstuhl für Security in Telecommunications

Compiler Assisted Vulnerability Assessment

vorgelegt von
M.Sc.
Bhargava Shastry
geb. in Davangere, Indien

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Ben Juurlink, Technische Universität Berlin
Gutachter:	Prof. Dr. Jean-Pierre Seifert, Technische Universität Berlin
Gutachter:	Prof. Dr. Marian Margraf, Freie Universität Berlin
Gutachter:	Prof. Dr. Konrad Rieck, Technische Universität Braunschweig
Gutachter:	Prof. Dr. Aurélien Francillon, EURECOM

Tag der wissenschaftlichen Aussprache: 14. September 2018

Berlin 2018

Ich versichere von Eides statt, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum

Abstract

With computer software pervading every aspect of our lives, vulnerabilities pose an active threat. Moreover, with shorter software development cycles and a security-as-an-afterthought mindset, vulnerabilities in shipped code are inevitable. Therefore, recognizing and fixing vulnerabilities has gained in importance. At the same time, there is a demand for methods to diagnose vulnerabilities within the software development process.

This dissertation introduces *compiler assisted vulnerability assessment*, a novel approach that brings together techniques from program analysis and software testing for recognizing vulnerabilities as software is written. The main idea behind our approach is to leverage compile-time analysis to abstract program information that is relevant for vulnerability assessment from source code. Our analysis is targeted at an in-memory code representation that exposes program syntax and semantics, making it both fast and precise. We show that our approach not only enables identification of well-studied vulnerability classes such as buffer overflows but also assists software testing and regression analysis. To this end, we develop three different methods for software vulnerability assessment, namely, a method to identify vulnerabilities spread across multiple source files, a method to infer the input format of parsing applications to test them more effectively, and a method to perform vulnerability template matching from fuzzer crashes. These methods address distinct tasks encountered by security practitioners on a regular basis. In doing so, they show how program-centric analyses and input-centric testing can complement each other towards vulnerability assessment.

We evaluate our methods on popular open-source software (OSS), quantifying their benefit in terms of number of new vulnerabilities exposed, time to expose vulnerabilities, and test coverage improvement. Our tools have found tens of zero-day vulnerabilities in production software such as Open vSwitch and tcpdump, and improved test coverage by 10–15%. In controlled settings, they have speeded up the time to vulnerability exposure by an order of magnitude. Our work shows that compiler assisted analysis has the potential to reduce the likelihood of vulnerabilities propagating to production software. More importantly, it shows that static program analysis holds promise for advancing the state-of-the-art in vulnerability assessment.

Zusammenfassung

Computersoftware durchdringt mittlerweile beinahe jeden Aspekt des alltäglichen Lebens—dies lässt Schwachstellen in Computersoftware zu einer aktiven Bedrohung werden. Darüber hinaus sind Schwachstellen durch kürzere Softwareentwicklungszyklen und durch Entwicklungsprozesse, in denen Sicherheit als nachträgliche und nebensächliche Adjustierung betrachtet wird, unvermeidbar. Daher hat das Erkennen und Beheben von Schwachstellen an Bedeutung gewonnen. Gleichzeitig besteht Bedarf an Methoden zur Diagnose von Schwachstellen innerhalb des Softwareentwicklungsprozesses selbst.

Diese Dissertation stellt Compiler-gestützte Schwachstellenanalyse vor—einen neuartigen Ansatz, welcher Techniken aus der statischen Programmanalyse und dem Software Testing kombiniert, um Schwachstellen noch während des Entwicklungsprozesses zu erkennen. Die Grundidee des Ansatzes ist hierbei, compile-time Analyse zu nutzen um diejenigen Informationen über das Programm zu extrahieren, die relevant für die Schwachstellenanalyse des Quellcodes sind. Ziel unserer Analyse ist eine in-memory Repräsentation des Codes, welche Syntax und Semantik enthält und dadurch sowohl schnell als auch präzise ist. Wir zeigen auf, dass unser Ansatz nicht nur die Analyse von bereits gut erforschten Schwachstellenarten wie zum Beispiel Pufferüberläufen ermöglicht, sondern auch Software Testing und Regressionsanalyse unterstützt. Hierzu entwickeln wir drei Methoden zur Schwachstellenanalyse: 1) Eine Methode um Schwachstellen, welche über mehrere Quellcode-Dateien verteilt sind, zu finden, 2) eine Methode zur Ermittlung des Eingabeformats von Parser Applikationen, wodurch diese effizienter getestet werden können und 3) eine Methode, um Vulnerability template matching von durch Fuzzing identifizierten Abstürzen durchzuführen. Diese Methoden helfen bei der Lösung von verschiedenartigsten Problemstellungen, welche von Sicherheitsexperten täglich angetroffen werden. Insgesamt zeigen die Methoden so, wie sich Programm-zentrierte Analysen und Eingabe-zentrierte Tests zum Zweck der Schwachstellenanalyse gegenseitig ergänzen können.

Wir evaluieren unsere Methoden anhand von populärer Open-Source-Software (OSS). Dabei quantifizieren wir ihren Nutzen hinsichtlich entdeckter Schwachstellen, der Dauer bis zur Entdeckung der Schwachstelle sowie der Verbesserung der Testabdeckung. Unsere Tools haben in Produktionssoftware, wie z.B. Open vSwitch und tcpdump, Dutzende von Zero-Day-Schwachstellen gefunden und die Testabdeckung

um 10-15% verbessert. In kontrollierten Testumgebungen haben sie die Zeit bis zum Aufdecken der Schwachstelle um eine Größenordnung verkürzt.

Unsere Arbeit zeigt, dass Compiler-gestützte Analyse das Potenzial hat, die Wahrscheinlichkeit des Verbleibens von Schwachstellen in Produktionssoftware zu verringern. Vor allem zeigt diese Arbeit, auch dass die statische Programmanalyse eine vielversprechende Technik ist, um den aktuellen Stand der Schwachstellenanalyse weiter auszubauen.

Publications Related to this Thesis

The work presented in this thesis resulted in the following peer-reviewed publications:

- *Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing*, **Bhargava Shastry**, Federico Maggi, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert, In the Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT), 2017. (see [144]/Chapter 5)

- *Static Analysis as a Fuzzing Aid*, **Bhargava Shastry**, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann, In the Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2017. (see [140]/Chapter 4)

- *Towards Vulnerability Discovery Using Staged Program Analysis*, **Bhargava Shastry**, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert, In the Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2016. (see [141]/Chapter 3)

- *A First Look at Firefox OS Security*, Daniel Defreez, **Bhargava Shastry**, Hao Chen, Jean-Pierre Seifert, In Proceedings of the Third Workshop on Mobile Security Technologies (MoST) 2014. (see [44]/Chapter 2)

The content of Chapter 3 has been adapted by permission from Springer Nature Customer Service Centre GmbH: Springer Detection of Intrusions and Malware, and Vulnerability Assessment Towards Vulnerability Discovery Using Staged Program Analysis, Bhargava Shastry, Fabian Yamaguchi, Konrad Rieck, Jean-Pierre Seifert, © Springer International Publishing Switzerland 2016. The content of Chapter 4 has been adapted by permission from Springer Nature Customer Service Centre GmbH: Springer Research in Attacks, Intrusions, and Defenses Static Program Analysis as a Fuzzing Aid, Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, Anja Feldmann, © Springer International Publishing AG 2017.

In addition, the insights gained from static program analysis and fuzz testing allowed the author to contribute to the following peer-reviewed publications:

- *Taking Control of SDN-based Cloud Systems via the Data Plane*, Kashyap Thimmaraju, **Bhargava Shastry**, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid, In the Proceedings of the 4th ACM Symposium on SDN Research (SOSR), 2018. **Best Paper Award.** (see [146])

- *The vAMP Attack: Taking Control of Cloud Systems via the Unified Packet Parser*, Kashyap Thimmaraju, **Bhargava Shastry**, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid, In the Proceedings of the ACM Cloud Computing Security Workshop (CCSW), 2017. (see [147])
- *Leveraging Flawed Tutorials for Seeding Large-Scale Web Vulnerability Discovery*, Tommi Unruh, **Bhargava Shastry**, Malte Skoruppa, Federico Maggi, Konrad Rieck, Jean-Pierre Seifert, and Fabian Yamaguchi, In the 11th USENIX Workshop on Offensive Technologies (WOOT), 2017. (see [151])

Acknowledgements

This dissertation has been shaped by the love and support of several people and serendipitous circumstances. My wife, Divya, and I arrived in Berlin with the hope that we can jointly advance our careers in research. My present research group happened to have two openings, one of which I filled. First year into my Ph.D. I realized that working in the chair for Security in Telecommunications—SecT for short—presents almost infinite freedom. I would like to thank my supervisor Jean-Pierre Seifert for providing an excellent research environment and freedom to pursue my research interests.

Ph.D. can get very intense. I am reminded of a quote attributed to both Antonio Gramsci and Romain Rolland, which is “I’m a pessimist because of intelligence, but an optimist because of will.” Being critical of one’s work is crucial to progress but it wears one out emotionally. Channeling our agency to make progress in the face of criticism requires self-belief. In times of intellectual angst, self-belief is in short supply. I would like to thank Divya for being my “motivational coach”, patiently instilling a sense of self-belief in me, and teaching me the value of not taking things too seriously. Thank you for giving me an ear and bringing me up whenever I was down, for explaining people dynamics and statistical concepts from time to time; without your love and support this dissertation would not have been possible.

Ph.D. is considered an individual achievement but it is far from it. We are “Stand[ing] on the shoulder of giants” to quote Google Scholar’s advertising slogan. The giants on whose shoulders I am standing are Jean-Pierre Seifert, Konrad Rieck, Fabian Yamaguchi and several other scholars in the field of computer security. I would like to thank Konrad Rieck for helpful discussions on inferring the input format of network parsers and on writing technical papers, Fabian Yamaguchi for helping me structure my contributions meaningfully and mentoring my vulnerability research work, and all my co-authors for supporting me in paper writing. Special thanks to Vincent Ulitzsch for patiently correcting a Google translated German version of the abstract of my dissertation and discussions on the Rice theorem. I have made several friends at SecT who I would like to thank. The conversations about research and life in Berlin during lunches with Kashyap Thimmaraju, Julian Fietkau, Felicitas Hetzelt and Robert Buhren played no small part in my Ph.D. A dissertation is incomplete without peer review and feedback. I would like to thank Marian Margraf, Konrad Rieck, and Aurélien Francillon for agreeing to be on my dissertation committee and

providing valuable feedback. I was fortunate not to worry about research funding. My research has been supported by the following grants: EU FP7 project NEMESYS (grant 317888), BMBF project Enzevalos (grant KIS1DSD032), and the API-Assistant activity under EIT Digital.

I would not have considered a career in research, were it not for the support and encouragement of my parents. I would like to thank my mother, Katyayini and my father, Suresh; I am grateful to them for making sacrifices for a brighter future for me.

Contents

To Ajji and Jhini

Introduction

1

> *Every program has (at least) two purposes: the one for which it was written and another for which it wasn't.*
>
> – Alan Perlis, *Epigrams on Programming*

We live in the so-called "Information Age" in which our reliance on computer systems is striking. Using digital services such as Internet banking and electronic commerce has become second nature to us. Computer infrastructure is the backbone of the modern economy, responsible for information storage, processing, and transmission at scale. Needless to say, we depend on the correct operation of computer systems. Software defects, or bugs, undermine our reliance on them. Of particular concern are software vulnerabilities, bugs that may be leveraged by a motivated adversary to cause intentional harm.

In our highly inter-connected world, vulnerabilities equip adversaries to disrupt day-to-day operations at a global scale. One of the earliest examples of this phenomenon is the Morris worm [115], which, in 1988, effectively brought down thousands of computers by exploiting, among others, a buffer overflow vulnerability in the *fingerd* network service. Three decades later, software vulnerabilities continue to expose digital infrastructure to widespread attacks. Indeed, the impact of present-day attacks is heightened by the high degree of interconnectedness of computer systems. Last year, the Wannacry worm [42] exploited a vulnerability in the Microsoft server message block (SMB) server implementation [99] to lock down machines with ransomware, bringing hospital services to a halt [70]. These incidents underscore the need for software vulnerability assessment.

Vulnerability discovery forms the basis of software vulnerability assessment. At its core, vulnerability discovery is a search problem whose aim is finding program defects *that provide leverage to attackers*. This distinguishes vulnerability discovery from bug discovery whose aim is finding program defects regardless of their utility to attackers. Therefore, a key concern of vulnerability discovery is reasoning about program behavior from an offensive perspective. Software vulnerability assessment is non-trivial because vulnerabilities are corner cases that escape attention under normal operating conditions.

Prior work addresses the software vulnerability assessment problem using three approaches that we will call input-centric, program-centric, and hybrid. Input-centric approaches develop run time mechanisms to monitor program behavior [135, 163, 108] and generate input that is likely to trigger vulnerabilities [56, 59, 117, 21, 57, 22]. Program-centric approaches focus on analyzing program syntax and semantics to uncover vulnerabilities in source code [152, 45, 89, 77, 159]. Hybrid approaches [142, 11, 101, 34, 153] combine both input and program centric approaches to assist in specific vulnerability assessment tasks such as checking input sanitization and generating vulnerability signatures. Although prior work has significantly advanced vulnerability discovery, diagnosis of vulnerabilities as software is written is a largely unaddressed problem.

In this dissertation, we address the problem of early vulnerability diagnosis. Our aim is to develop vulnerability assessment tools that can be integrated into the software development lifecycle so that vulnerabilities are detected before software is shipped to end-users. Our approach brings together concepts from program analysis and testing while benefiting from advances in compiler technology. In particular, we present program analysis methods that are *adaptable* to both program and input centric paradigms and *practical* enough to discover vulnerabilities in production software. In the rest of this chapter, we introduce the reader to the task of vulnerability assessment and ideas from program analysis. Subsequently, we briefly describe our approach and proceed to highlight the primary contributions of this dissertation. We conclude this chapter with an outline of this thesis, briefly summarizing the content of each chapter. We discuss related work separately in each of the following chapters.

1.1 Vulnerability Assessment

Vulnerability assessment is a recent addition to computer security research. In contrast to intrusion detection [5], vulnerability assessment is concerned with the discovery of software vulnerabilities even before they may manifest as attacks. The Internet Security Glossary (IETF RFC 4949) [111, page 333] defines a vulnerability as follows.

Definition 1.1. Vulnerability is a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.

Security policies usually center around one or more of the following security requirements: maintaining *confidentiality*, *integrity*, or *availability* [15], colloquially known as the C-I-A triad. Thus, vulnerabilities are system defects that permit violation of the C-I-A triad. We make two qualifications to Definition 1.1 to present the scope of our work. First, we exclude vulnerabilities in the operational and management aspects of a system so that we can focus on vulnerabilities in its design and implementation. Second, we consider vulnerabilities in computer software only, excluding vulnerabilities in the hardware and firmware components of a typical computer system. This permits us to assess vulnerabilities in a computer program in isolation.

The definition of vulnerability assessment follows Definition 1.1.

Definition 1.2. Vulnerability assessment is the systematic examination of a system to identify vulnerabilities.

Central to vulnerability identification is assessing attacker leverage. The C-I-A triad provides a basis for assessing the impact of a vulnerability at a policy level. However, the empirical impact of a vulnerability is judged by the attacker's ability to perform one or more of the following malicious actions.

- **Execute arbitrary code.** An attacker may execute code of their choice on the victim's system. For example, they may leverage vulnerable code to install a back door in a networked system.
- **Disclose information.** An attacker may steal private data on a victim system. For example, they may exercise vulnerable code to steal victim's cryptographic private key.
- **Deny service.** An attacker may disrupt a widely-used service. For example, they may exercise vulnerable code to repeatedly shut down a web service or cause it to be flooded with requests so that it is effectively shut down.

For example, Listing 1.1 shows a code snippet from Open vSwitch containing an exploitable buffer overflow vulnerability that was discovered by the author. The vulnerability is in the function called `parse_mpls` that is responsible for parsing MPLS packets sent to the switch from a remote end-point. `parse_mpls` accepts a pointer to raw packet data `datap` as input and returns the number of MPLS labels present in the packet as output (line 14). It does so by incrementing a local counter for each parsed label until it encounters a label that has a set bit in the bit position reserved by the MPLS standard for the last MPLS label [110]. `parse_mpls` then returns the number of MPLS labels to the calling function `miniflow_extract` that

```
 1  /* Pulls the MPLS headers at '*datap' and returns the count of them. */
 2  static inline int
 3  parse_mpls(void **datap, size_t *sizep)
 4  {
 5      const struct mpls_hdr *mh;
 6      int count = 0;
 7
 8      while ((mh = data_try_pull(datap, sizep, sizeof *mh))) {
 9          count++;
10          if (mh->mpls_lse.lo & htons(1 << MPLS_BOS_SHIFT)) {
11              break;
12          }
13      }
14      return MAX(count, FLOW_MAX_MPLS_LABELS);
15  }
16
17  /* Caller is responsible for initializing 'dst' with enough storage for
18   *  * FLOW_U32S * 4 bytes. */
19  void
20  miniflow_extract(struct ofpbuf *packet,
21                   const struct pkt_metadata *md,
22                   struct miniflow *dst) {
23
24      void *data = ofpbuf_data(packet);
25      size_t size = ofpbuf_size(packet);
26      uint32_t *values = miniflow_values(dst);
27      struct mf_ctx mf = { 0, values, values + FLOW_U32S };
28
29      ...
30
31      /* Parse mpls. */
32      if (OVS_UNLIKELY(eth_type_mpls(dl_type))) {
33          int count;
34          const void *mpls = data;
35          count = parse_mpls(&data, &size);
36          miniflow_push_words(mf, mpls_lse, mpls, count);
37      }
38  }
```

Listing 1.1: Buffer overflow in Open vSwitch that permits remote code execution.

is responsible for copying the labels from the packet stream to a local data structure called `dst` that is stack allocated (line 36).

However, not accounting for the fact that the input to the `parse_mpls` function (packet data) is attacker controlled, it returns a maximum of the number of parsed labels or a constant that denotes an upper bound on the number of MPLS labels (see line 14, `FLOW_MAX_MPLS_LABELS`). Should the attacker send an Ethernet packet that contains more than `FLOW_MAX_MPLS_LABELS` MPLS labels, the number returned by `parse_mpls` leads to a stack buffer overflow in `miniflow_extract`. This vulnerability has been shown to be quite severe, permitting a remote attacker to take full control of data center infrastructure [146].

It is evident from our discussion that reasoning about program behavior, especially in corner-cases such as this, is central to recognizing vulnerabilities and gauging

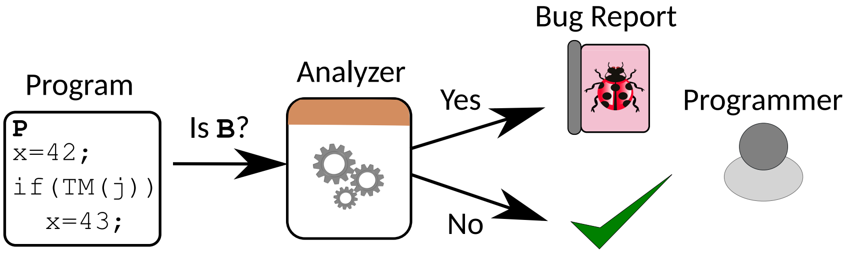


Fig. 1.1.: Program analysis as a decision procedure for vulnerability identification.

their impact. Program analysis, a field of study that is focused on the automatic analysis of program behavior, is well-suited for this task.

1.2 Program Analysis

Program analysis is concerned with the task of developing methods and techniques for analyzing other programs. Typically, program analysis targeted at bug detection is a decision procedure for the following question: Given source code of a program `P` and an undesired program property `B`, does `P` exhibit `B` in at least one of all feasible executions. If the answer to this question is yes, a bug report detailing the violation is presented to the programmer. Figure 1.1 illustrates this setting.

There are two flavors of program analysis: dynamic and static. Dynamic program analysis is the study of algorithms that reason about program behavior during its execution. This field of study [14, 79, 155, 112, 135] has many useful applications such as understanding program behavior and testing it for correctness. Dynamic program analysis serves as a useful debugging aid, especially for vulnerability assessment, because program behavior may be observed for *specific* executions. However, because it is execution (and hence input) centric, observable program behavior is limited by the number of distinct program inputs provided.

Static program analysis is the study of algorithms that reason about program behavior without executing the program. This field of study is useful not only for designing optimizing code compilers [3] but also tools that assist in software development, including but not limited to software vulnerability assessment [80, 62, 33, 51, 67]. Although the application of program analysis for reasoning about program properties is highly desirable, all non-trivial [1] program properties are undecidable [130]. Even though this appears to be a discouraging result, it does not detract from the utility

---

[1] A non-trivial program property is one that is true for at least one program and false for at least one program.

of program analysis for solving practical problems such as making the program run faster and finding vulnerabilities.

Since reasoning about non-trivial program properties is undecidable in general, a program analyzer may make one of the following three choices:

- **Unsound**: May conclude program is *buggy* even when it is not
- **Sound but incomplete**: May conclude program is *not* buggy even when it is
- **Non-terminating**: Is sound and complete but may never terminate

Naturally, a non-terminating vulnerability detector is unappealing because programmers desire quick actionable advice. A sound vulnerability detector is desirable because all flagged vulnerabilities are indeed real. For example, a fuzz testing tool such as *afl-fuzz* is sound with regard to severe memory corruption vulnerabilities because all program crashes discovered by it are caused by verifiable vulnerabilities. However, sound tools inherently miss vulnerabilities, demanding complementary techniques to increase confidence in the program. From a static program analysis perspective, designing a sound vulnerability detector is notoriously difficult because of two reasons. First, static analysis can not anticipate dynamic program properties such as a memory allocation that is dependent on program input, because of which it may miss real vulnerabilities. Second, static analyses approximate program behavior to be tractable, making false negatives inevitable. Despite these constraints, *even* unsound static program analysis can complement sound vulnerability detectors such as fuzzers. There is always scope to design a static analyzer that makes more precise approximations to cater to practical problems, a principle that is colloquially called the full employment theorem for static program analysis designers.

Next, we introduce concepts central to static program analysis in general and vulnerability identification in particular. First, we discuss different stages of program analysis, namely, lexical, syntax, and semantic analysis. Second, we discuss data and control flow analysis that are fundamental to understanding the flow of data and program statements.

1.2.1 Analysis Stages

Lexical Analysis

Lexical analysis is one of the simplest forms of program analysis whose purpose is to decompose the input program into chunks for subsequent analysis. Lexical analysis accepts program source code as input and parses the program into sequences called *lexemes* that hold meaning in the underlying programming language. Lexemes are then tokenized such that each lexeme corresponds to a token. Although lexical analysis is extremely fast, it provides little structural insight into the program. The ITS4 scanner [152] is one of the earliest examples of a vulnerability analyzer based on lexical analysis.

Syntax Analysis

Syntax analysis, also called parsing, takes tokens produced by lexical analysis as input and produces a parse tree that encapsulates the underlying language grammar. The parse tree is also called the abstract syntax tree (AST) since at this stage concrete syntax such as variable names have been abstracted away. Syntax analysis is also extremely fast, and although it provides structural insight into the program it lacks an understanding of program semantics. The concept of vulnerability extrapolation [160] is a good example of the use of abstract syntax for vulnerability identification.

Semantic Analysis

Semantic analysis takes the abstract syntax tree of the program as input and checks that the program conforms to language semantics. For example, *type checking* is a semantic analysis that checks that each operator in the language definition (such as addition) have operands of the same type. Semantic analysis is slower compared to syntax analysis because it requires making multiple passes over a program abstraction such as the control flow graph [31]. Livshits et al. [89] use a semantic analysis technique called points-to analysis to find security vulnerabilities in Java code.

1.2.2 Analysis Types

Data-Flow Analysis

Data-flow analysis refers to a collection of techniques that reason about the flow of data in the program. Data-flow analysis is used by compilers towards code optimization. For example, *constant propagation* is a data-flow analysis that seeks to replace expressions that always evaluate to a constant value by the constant.

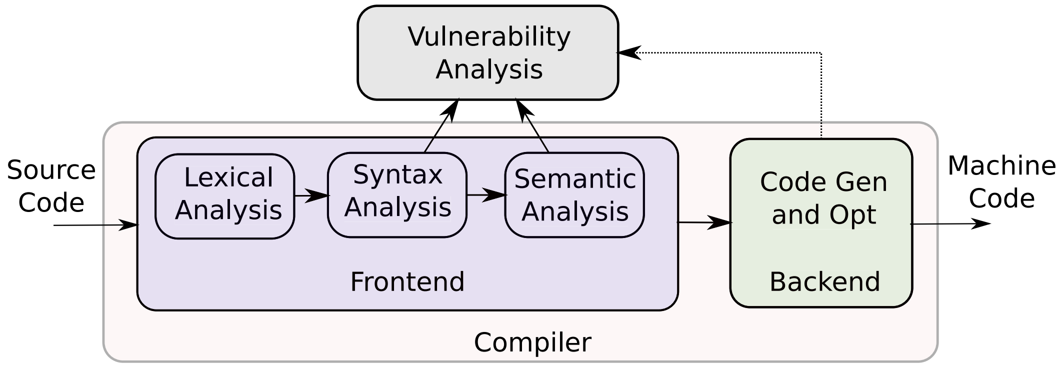


Fig. 1.2.: High level structure of a modern optimizing compiler depicting the placement of our vulnerability analyzer.

Taint analysis [89, 133] is another example of data-flow analysis that is of particular importance for vulnerability assessment. Taint analysis analyzes the flow of data between two program points of interest: a *source* of potentially attacker controlled data and a *sink* that performs some security sensitive operation.

Control-Flow Analysis

While data-flow analysis reasons about the flow of data in the program, control flow analysis reasons about the flow of program statements. For example, a conditional program statement such as `if` can have two outcomes, one in which the statement in the scope of the `if` statement is executed and the other in which the program statement following the `if` statement is executed. Compilers use the control flow graph (CFG) data structure to reason about control flow. The CFG is a directed graph in which nodes represent program statements and edges represent program path chunks. Control flow analysis is essential to vulnerability identification because by reasoning about the feasibility of program paths, it can tell us how a vulnerability may be triggered.

1.3 Compiler Assisted Vulnerability Assessment

As we have seen, program analysis is well-suited for the task of vulnerability assessment. However, performing early vulnerability diagnosis poses two challenges. First, the task of vulnerability assessment should integrate with modern software development lifecycle, otherwise programmers and testers will not get timely feedback.

Second, since modern software is dynamic and is modified on a daily basis, vulnerability analysis should be reasonably fast so that it may be performed continuously. We require an analysis platform that addresses both these challenges while enabling precise vulnerability diagnosis.

A compiler is an ideal analysis platform for this setting because it is not only a standard component in the software development workflow but also permits fast analysis. Indeed, compilers may be considered the most basic vulnerability analyzers that point out defects in code. Naturally, extending the compiler framework to perform more targeted vulnerability analysis is appealing. This is the key idea behind our approach that we call *compiler assisted vulnerability assessment*. We use the name to collectively refer to any form of compiler-based program analysis that is useful for vulnerability assessment.

Figure 1.2 shows the architecture of a modern optimizing compiler and the placement of our vulnerability analysis. As shown in Figure 1.2, a compiler is a program that accepts source code as input and produces machine code as output. It may be divided into two components: the front-end and the back-end. The former is responsible for parsing source code and performing syntax and semantic analysis, while the latter is responsible for generating optimal machine code.

The primary task of the compiler front-end is to check that the source program is well-formed and understand program semantics toward code generation and optimization. This requires the compiler to perform multiple stages of analysis to first understand the meaning of program statements before attempting to produce a more optimal version of the program. Specifically, well-formedness is checked using lexical and syntax analysis and optimization is performed based on the outcome of semantic analysis. Program insight gained through syntax and semantic analysis performed by the compiler is also useful for vulnerability analysis because vulnerabilities are ultimately a program property. Although the compiler back-end is primarily concerned with code optimization, modern compiler back-ends such as LLVM [85] permit whole program analysis that help diagnose vulnerabilities spread across source files. Therefore, our analysis is modeled as plug-ins to the syntax and semantic analyzers of a modern compiler front-end, occasionally taking advantage of the global scope of a compiler back end for performing more precise analysis.

Before we can undertake the design and implementation of a vulnerability analyzer, we need to understand program properties that characterize vulnerabilities. This is the motivation behind the next chapter. The design space of compiler-based techniques for vulnerability identification is constrained by the following factors.

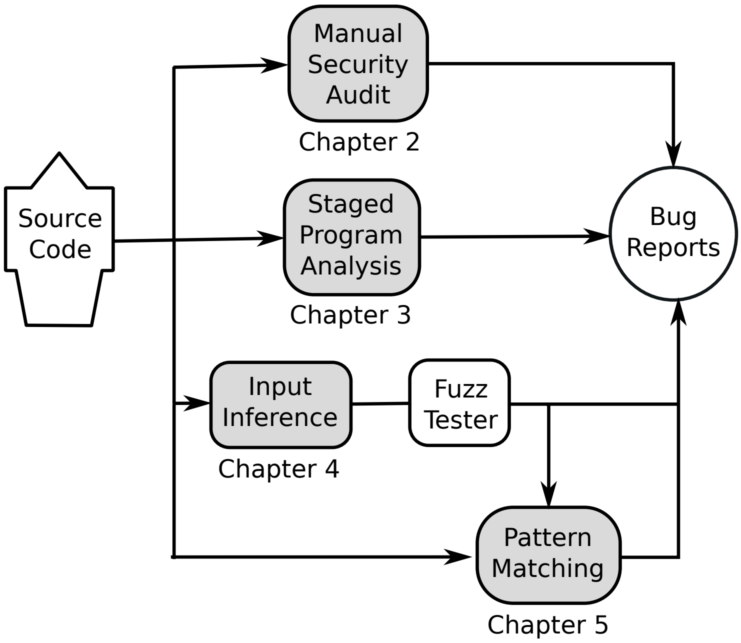


Fig. 1.3.: A schematic illustration of our vulnerability discovery process and core contribution. Gray blocks are our novel contribution.

First, these techniques should address clear-cut tasks that are useful in the vulnerability diagnosis process. Second, they need to mesh together with the software development workflow. These factors influence the design and implementation of vulnerability assessment methods presented in Chapters 3–5.

1.4 Thesis Contribution

In this dissertation we address the problem of vulnerability assessment of open-source software by combining concepts from program analysis and testing. We present a program analysis framework based on a modern compiler infrastructure for discovering memory corruption vulnerabilities in large object-oriented codebases and network parsers.

Figure 1.3 shows a schematic overview of our contribution. Broadly speaking, our proposed methods accept source code of program under analysis as input and produce vulnerability reports as output. They may be applied at compile-time (Chapter 3), prior to fuzz testing (Chapter 4), post fuzz testing (Chapter 5), or post production (Chapter 2). The following contributions have made our work possible.

- **Security Audit of Firefox OS**: We perform the first security audit of Firefox OS, an emerging web based smartphone OS, and uncover vulnerabilities that manifest at the intersection of the web and smartphone paradigms. This work serves as a stepping stone to understanding the requirements of and designing a vulnerability analyzer (Chapter 2).
- **Diagnosis of Distributed Vulnerabilities**: We introduce a novel paradigm for analysis of vulnerabilities that manifest across multiple software components in large codebases, and demonstrate that our analysis is capable of flagging real vulnerabilities. Our analysis comprises two stages to diagnose vulnerability classes such as type confusion and use of uninitialized variable in large object oriented codebases (Chapter 3).
- **Static Analysis Guided Fuzzing**: We propose the use of static program analysis as a fuzzing aid, demonstrating that program analysis may be flexibly used for vulnerability assessment. To enable this coupling, we leverage program analysis to infer the format of network packet and file formats and feed the resulting information to the fuzzer. We show that program analysis guided fuzzing is better than baseline fuzzers in terms of test coverage, time to vulnerability exposure, and the number of vulnerabilities uncovered (Chapter 4).
- **Static Vulnerability Template Matching**: We develop a method to find replicas of zero-day vulnerabilities found by fuzz testing, leveraging this method to identify regressions in a production networking switch. To this end, we propose a pipeline for automatically constructing vulnerability templates from fuzzer crashes, and match them against the code base. This has helped identify not only software regressions but new bugs that were in untested code paths (Chapter 5).

We have open-sourced the tools developed during this thesis, namely *Mélange* [138], *Orthrus* [139], and *afl-sancov* [137] under a GPL license. These tools have helped diagnose and fix multiple vulnerabilities in open-source software.

1.5 Thesis Organization

This thesis consists of six chapters and an appendix. The first chapter serves as a general introduction. In Chapters 2–5, we present our methods for vulnerability assessment, their evaluation, and a discussion of related work. These chapters have been written so that they may be read independently of each other. However, reading them in sequence is recommended. Chapter 2 emphasizes manual methods while Chapters 3–5 emphasize semi-automated methods for vulnerability assessment. Chapter 6 concludes this dissertation. Appendix A lists zero-day vulnerabilities that were discovered and subsequently reported during the course of this dissertation. In the following, we briefly summarize the content of the remaining chapters.

Chapter 2 We start with a brief introduction to web-based smartphone OS in general and Firefox OS in particular. We proceed to perform a security audit of Firefox OS using a penetration testing approach and discuss our findings. This work serves as a motivation to develop tools for vulnerability discovery.

Chapter 3 We begin an investigation of the utility of compiler-based analysis for vulnerability discovery. Focusing on the problem of identifying vulnerabilities spread across source files and motivated by real-world vulnerabilities, we describe the design and evaluation of a staged program analyzer that we call Mélange.

Chapter 4 We continue our discussion on compiler-based vulnerability assessment by focusing on the problem of testing parsing applications. Motivated by the challenges specific to testing parsing applications whose behavior is heavily influenced by the supplied input, we describe the design and implementation of a static analysis guided input inference engine that may be used to assist fuzz testing tools. We demonstrate that this inference can improve program testing, leading to the discovery of new vulnerabilities and increasing test coverage in production code.

Chapter 5 We briefly describe the challenges in performing regression testing, another important task in vulnerability assessment. We introduce the reader to static template matching and its application to regression analysis. We show that coupling static analysis and fuzz testing is well-suited to tackle the problems of regression analysis and the detection of recurring zero-day vulnerabilities. We evaluate our method using a case study of a popular networking switch. This chapter is the last of the three methods for compiler-based vulnerability assessment.

Chapter 6 We conclude this dissertation and present a summary of work carried out. Finally, we present avenues for future work in the field.

Appendix A We list zero-day vulnerabilities found during the course of this dissertation.

2

A First Step in Vulnerability Assessment

Smartphones have changed how we interact with the digital world, making services available at our finger tips. Providing services via mobile applications—the mobile application provisioning model—is the accepted norm today. This provisioning model relies largely on a portable systems application programming interface (API) so that applications can be written once and run on diverse hardware platforms. Owing to its portability and maturity as a programming language, Java is a popular choice for application development in the Android ecosystem.

In recent times, the idea of leveraging a web backend for smart devices has received attention [72, 116, 93]. Mozilla's efforts towards a web-based smartphone operating system—called Firefox OS [124]—is a prominent example. Firefox OS comprises a web backend—a JavaScript interpreter and a rendering engine—engineered on top of the Linux kernel. Firefox OS applications written in HTML/JavaScript and CSS, may be either hosted remotely on a server or packaged and hosted in a central application store.

In this chapter, we perform a *security audit of Firefox OS* using manual source code audits and a penetration testing approach [154]. This complements the static analysis approach taken by Mozilla for Firefox OS applications [105]. Moreover, our approach is well-suited to answer specific research questions such as: "How secure is the enforcement of the same origin policy in Firefox OS?" and "Is TLS certificate validation performed correctly?" Indeed, our goal is to understand specific security challenges that arise at the intersection of mobile and web technologies.

The rest of this chapter is organized as follows. We begin by providing background information on the Firefox OS architecture. Next, we briefly motivate the task of performing a security assessment of Firefox OS. Subsequently, we present our study methodology and discuss results of the study. This is followed by a discussion of related work. We conclude the chapter with a discussion on the need for automated and more scalable techniques for performing security audit of application and systems software.

2.1 Background

Firefox OS is a web operating system that comprises three components called Gonk, Gecko, and Gaia. Gonk is similar in spirit to the Linux kernel used in the Android stack. It abstracts the underlying hardware, providing system services such as process management, privilege separation etc. Gecko comprises Mozilla's JavaScript interpreter (SpiderMonkey) and the web rendering engine that is responsible for displaying HTML/CSS applications. Applications written in HTML/JavaScript/CSS are run by Gecko. Finally, Gaia comprises the UI of Firefox OS. It consists of a set of applications that are shipped with Firefox OS devices.

2.1.1 Types of Applications

Firefox OS applications can be classified by their privilege level as follows: unprivileged, privileged, and certified. Unprivileged applications can make use of a restricted set of device permissions. They may be hosted on a remote server or packaged. In practice, the installation of hosted applications is akin to bookmarking a web page; the installed application only serves as a shortcut to the hosted content. On the other hand, a packaged application comprises the software required to run and render it on the device. The software (HTML/JavaScript/CSS files along with an application manifest) are packaged in a zip file that is distributed via an application marketplace similar to the Android app store. The marketplace is responsible for signing all packaged applications. The application manifest contains meta-data about the application including an exhaustive list of device permissions requested by the application.

Privileged applications, as their name suggests, may make use of device APIs that are considered security sensitive. Some examples of security sensitive APIs include geolocation, contacts etc. Privileged applications must be packaged and conform to the default content security policy (CSP) defined by Mozilla. Privileged applications are reviewed before being accepted into a marketplace. Finally, certified applications are those applications that are bundled together with a device. Certified applications have access to all permissions.

2.1.2 Application Distribution

The distribution and installation of unprivileged applications depends on whether they are hosted or packaged. Unprivileged packaged applications are distributed via an application store in the form of a `zip` archive. The zip archive is digitally signed by the app store and delivered over HTTPS. This ensures the integrity of application code. On the other hand, hosted applications are distributed directly by the application developer via a hosting space of their choice. Although hosted applications function like bookmarks, their installation involves fetching an application `manifest` file over the network. The manifest file contains meta-data about the application and the permissions requested by it.

2.1.3 Device Permissions

Like Android, Firefox OS uses a permission model to provide access to web applications. There are a total of 56 permissions available of which only 23 are usable by uncertified (i.e., hosted) applications. Firefox OS permissions may be either implicit or explicit. Implicit permissions are those permissions that are automatically granted if they are present in the application's manifest file. On the other hand, explicit permissions trigger a user prompt at first use, and more importantly are revocable at any time via the `Settings` application.

2.2 Task: Firefox OS Security Audit

Web security has matured over the years. The web has been exposed to large scale attacks [95, 92] for a long time. Defensive techniques, such as the use of transport level security (TLS) and cross-site scripting (XSS) filters, have evolved as a response to these attacks. There is an expectation that a web-based smartphone will inherit the defensive outlook of the web. In this chapter, our aim is to empirically assess if this expectation is justified or not. Firefox OS is a natural target for our case study since it is one of the first web-based smartphones targeted at a mass user base [124].

Firefox OS is massive code base comprising over 15 million source lines of code. We constrain the scope of our study because a complete security audit at this scale is resource intensive. Since Mozilla performs static analysis of applications admitted to the application store [105], we focus on auditing dynamic application behavior on the one hand, and applications' interaction with the underlying operating system on the other. Therefore, our task complements existing security measures adopted by Mozilla.

Security audit of Firefox OS poses two challenges. First, given the massive codebase, reviewing each and every line of code is not suitable. Therefore, we need to select portions of code that contribute to system security. Second, an experimental set up needs to be established to facilitate monitoring of dynamic program behavior. To address these challenges, we use a combination of manual source code analysis and penetration testing [154], an approach favored by security practitioners for assessing system security.

2.3 Penetration Testing Firefox OS

Weaknesses in the smartphone permission model [48, 47, 17] and TLS certification validation in smartphones [46, 55] have received significant attention in the security community. Taking a cue from past work, we focus attention on portions of Firefox OS that relate to application installation and permission checks and TLS certification validation. Using manual source code audit, we note that application installation in Firefox OS is handled by the boot2gecko process, while TLS certificate validation is performed by the network security services (NSS) library developed by Mozilla. In the following, we treat these subsystems as targets of our analysis.

To perform our task, we set up an experimental test bed that consists of the following components: (1) The Geeksphone Peak [54] smartphone running a pre-release version of Firefox OS version 1.4.0.0; (2) Firefox OS simulator [104], the OS emulator developed by Mozilla; (3) MITMProxy [32], a popular proxy application for performing person in the middle attacks. We describe our findings in the following paragraphs.

In particular, we describe what happens when system services that used to be tightly coupled to the browser program are exposed to applications in a mobile environment without modification. Alongside insufficient security UI in Firefox OS, we demonstrate how the SSL/TLS certificate caching problem poses a security risk. Furthermore, we bring attention to the use of plain text communications for provisioning applications, and show that an attacker can inject new functionality (and permissions) during the application download process.

2.3.1 SSL Certificate Caching

The certificate caching problem—a known issue in Firefox OS [18]—is the following: Manually overriding a certificate warning for a web origin not only overrides the warning for subsequent visits to that web origin, but also makes the override applicable to any application on the phone, as though applications queried a shared system-wide cache for certificate overrides. Since application code from the same web origin is sandboxed across applications in Firefox OS, the handling of overrides presents an anomaly. We digress to present a simplified flow of SSL certificate validation and override in Gecko. Subsequently, we analyze the problem of certificate caching in detail.

Certificate Validation and Override Services Gecko relies on the Network Security Services (NSS) library [109] for security related services. NSS' SSL/TLS certificate validation service is responsible for validating the SSL certificate chain presented by a remote server in the process of initiating an HTTPS connection. The certificate override service, tied to the certificate warning interstitial page, allows for manual overrides of certificate warnings. It maintains a record of certificate overrides that have taken place in the past, in the form of {Web Origin, Certificate Fingerprint} key-value pairs. For a temporary override, the key-value pair is cached in Gecko's program memory. When certificate validation for a server fails, the validation service queries the override service to check if an overridden certificate for the server has been cached. If there is a cache hit, the override service verifies if the cached certificate is the same as the certificate presented in the ongoing SSL handshake. If this succeeds, the HTTPS connection proceeds without a warning. Otherwise, the user is presented with a certificate warning interstitial.

A valid certificate vouches for the authenticity of a remote security principal and forms the basis for trust in Internet communications. The browser web and the smartphone platform entertain different notions of security principals. Next, we see how this leads to a confused deputy scenario in Firefox OS.

Security Principal Desktop browsers routinely interact with off-device security principals. The same origin policy establishes a means to identify them. While the policy is reflected in legacy NSS code, the notion of an on-device security principal is something that is alien to security services in the NSS library in general, and the certificate validation and override services in particular.

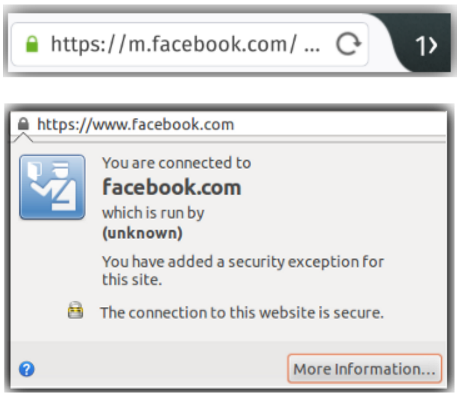
While Firefox OS relies on the underlying OS kernel for isolating on-device principals, it delegates certificate validation and overriding to the legacy NSS library. So, while the application is treated as a security principal in managing session tokens, local storage, and permission-based access control, the web origin dictates how SSL certificates are handled. This semantic difference between the NSS port in the desktop browser and in Firefox OS makes both the certificate validation and override services, confused deputies. Since the two services still treat web origin as the security principal, certificate validation requests for the same web origin *across* Firefox OS applications elicit the same response.

Next, we examine architectural differences between the desktop browser and Firefox OS that lead to side-effects being distributed and subsequently retained across process boundaries.

Process Model The desktop browser and Firefox OS are built on different process models. The desktop browser program itself and all of its (web) content run in a single OS process. In contrast, Firefox OS is built on a multi-process model. Each web app on Firefox OS runs in a separate *Content* process. Gecko and its constituent sub-systems run in a privileged process called the boot2gecko (b2g) process.

Because the desktop browser and its content pages run in a single process, all transient side-effects are contained within the process' memory; this includes temporary certificate overrides. On Firefox OS web apps running in content processes are untrusted principals, but the b2g process belongs to the trusted computing base (TCB). Security sensitive side-effects such as certificate overrides are registered in the b2g process, even if the operation responsible for the side-effect originated in a content process. The validity of the override stretches through the lifespan of the b2g process. Given that the b2g process is one of of Firefox OS' core components, it is killed only on device restart.

A defining characteristic of the desktop user experience is users being able to start and close (kill) programs (OS processes). This observation combined with the fact that desktop users are exposed to the browser program in its entirety (program = process) instills the notion that side-effects are strictly tied to program lifespan. The smartphone user experience on Firefox OS is starkly different. Since program (process) management of the b2g process is not exposed to smartphone users, user expectations around side-effects are not respected by the underlying OS.



https://m.facebook.com/ ...

https://www.facebook.com

You are connected to

facebook.com

which is run by

(unknown)

You have added a security exception for this site.

The connection to this website is secure.

Fig. 2.1.: Site identity button after a certificate override: (Top-Bottom) Firefox OS browser, Firefox desktop browser

Put together, application agnostic certificate handling and retention of overrides in a core process pose a security risk. Next, we see how the problem is exacerbated by discrepancies in the security UI of Firefox OS.

UI Discrepancy Although the caching of temporarily overridden certificates is a characteristic of the desktop browser, there are checks in place to ensure that an active attacker has limited leverage. Temporary certificate overrides are removed when the user closes the browser program. However, a browser restart is not a necessary prerequisite to clear overrides. Desktop Firefox exposes the option[1] to remove overrides during a running instance of the program. Furthermore, even if overrides are not cleared, information about past overrides is captured in multiple visual security indicators in the browser UI; these indicators assist users in taking informed decisions. One such indicator is a UI element called the *site identity button* [102]. The site identity button is comprised of a padlock icon at a bare-minimum. Figure 2.1 shows comparative screen shots of the site identity buttons in the Firefox OS browser and the Firefox desktop browser after a certificate override has taken place. As shown in the figure, the padlock icon is gray colored when a certificate override has taken place on the desktop browser. On clicking the padlock, an informative security dialog is displayed to the user which presents textual feedback on the state of connection security. While the browser app on Firefox OS has options which permit clearing browsing history, cookies, and stored data, none of these are tied to sanitizing temporary certificate overrides. An examination of the source code of Firefox OS' NSS port reveals that the option of overriding `Active`

---

[1] Removal of temporary certificate overrides is tied to clearing Active Logins from recent browsing history.

`Logins` is present. Evidently, this option is neither exposed in the browser app nor the certified Settings app.

The site identity button in Firefox OS' browser app has the following states:

1. Web page has no security (HTTP), visually represented by the globe icon,
2. Page is insecure or displays mixed (HTTP and HTTPS) content, represented by a broken gray colored padlock,
3. Page is secure, represented by a green colored padlock.

In the Gecko port for Firefox OS, certificate overrides are incorrectly mapped to the secure state of the site identity button; the padlock is thus green colored (see Figure 2.1) even when a web page's SSL certificate has been manually overridden. Additionally, because of limited screen real estate, the security information dialog that is supposed to pop-up on clicking the site identity button is absent in the browser app. In the mozbrowser view that is commonly used by Firefox OS apps to render web pages, the site identity button is absent. Certificate validation or overrides in the mozbrowser view therefore go unnoticed. Given that temporary certificate overrides remain in Gecko's program memory for a substantial time-period, this poses a realistic threat. Tricking the user into overriding the certificate warning for a given web origin is sufficient to compromise subsequent visits to the affected web origin across multiple apps. This includes the default browser app, *mozbrowser* instances of apps belonging to the same domain, and third-party apps that load web content (e.g. for OAuth based authentication) from the affected web origin. This was manually verified in our experimental testbed.

Threat Scenario

1. Victim connects to compromised network, such as public wifi.
2. Attacker performs MITM attack against HTTPS[2] and presents a fake certificate.
3. Victim overrides certificate warning temporarily, assuming the action impacts only the present page visit.
4. Page is loaded with green padlock in the location bar.
5. All subsequent connections can be MITM.

---

[2] Web domains for which HTTP Strict-Transport-Security (HSTS) is hard-coded in the browser are not vulnerable to the described attack primarily because certificate overrides for these domains are disallowed.

If device restart has not taken place, the MITM attack outlined in the preceding scenario might recur should the victim reconnect to the compromised wifi at a later time. On successive connections to a domain whose certificate has been overridden by the victim in the past, a full SSL handshake with the attacker's cached SSL certificate takes place silently i.e., the victim is neither presented with a warning interstitial, nor is a possible MITM signaled in the browser UI. Apart from initiating MITM attacks on newer connections, the adversary can steal cookies from the victim's unexpired sessions with the compromised domain across applications.

A malicious application could aid this process by detecting when a MITM attack is present. The application could provide the user with some plausible sounding reason for requiring a certificate override. Certificate warnings are already confusing, and any inconsistencies in sandboxing behavior only increase the cognitive attack surface.

2.3.2 Application Code and Manifest Provisioning

While Mozilla ensures that packaged applications are digitally signed and fetched from the marketplace over SSL, hosted applications are provisioned as if they were normal web content. Because hosted apps still need to fetch application manifests and code from a web origin, Mozilla recommends that app developers serve them over HTTPS [103].

The application manifest file contains security sensitive information including a list of permissions requested by an app among other things. We collated an exhaustive list of hosted manifest URLs and noted that 92.8%[3] of hosted applications fetch manifest and application code over HTTP. We evaluated the threat of a man-in-the-middle intercepting a hosted manifest/application code and subsequently modifying it before relaying the modified content to the end-user. As proof of concept, we inserted the *audio-capture* (microphone) permission in the manifest of an application that is designed to only *play* audio content. Subsequently, while application code was being fetched from the remote server (over HTTP), we altered it by adding a record audio functionality to the app.

The threat of an active attacker modifying the manifest and application code in transit is constrained in two ways. Firstly, sensitive permissions available to hosted apps such as *audio-capture* and *geolocation* are explicit i.e., prompt on first use. Secondly, Mozilla segregates permission sets for hosted and privileged applications.

---

[3] 1106 out of a total of 1191 hosted applications returned by the marketplace API, as of 22nd February, 2014

Should an attacker request, say, the *Contacts* permission (a privileged permission on Firefox OS), Gaia's application installer component will trigger a signature check. Upon finding that the file in question is a manifest and not a signed zip package, the installation process is aborted. Unprivileged implicit permissions, however, can be injected without the user's knowledge. Since the *App Permissions* tab in the *Settings* app does not list implicit permissions, there is no way for a user to tell that the permission requests have been modified. Presently, implicit permissions for hosted apps include *audio*, *fmradio*, *alarms*, *desktop-notification*, and *storage* [106]. These are relatively benign permissions, but their absence from the permissions UI combined with the lack of an integrity mechanism for hosted application code would be a cause for concern should Mozilla enlarge the set of implicit permissions for hosted applications.

We also observed that several popular Firefox OS applications such as Free SMS [134] encode user identifiable information (including phone numbers) in URLs and communicate over HTTP. Moving forward, requiring that security sensitive hosted applications serve their manifest (and application code) over HTTPS would be a step in the right direction.

2.4 Related Work

To the best of our knowledge, we are the first to look at the security of Firefox OS. The differences between Android and Firefox OS permission enforcement are discussed in [75], but the focus of that work is on Android. While real-world studies [55, 46] have observed problems in SSL validation across popular web and smartphone applications and middleware libraries, they do not look at how certificate overrides are handled at the client side. On Firefox OS, SSL validation is done by system software; applications are not trusted to validate certificates on their own.

Amrutkar et al. [4] perform an empirical evaluation of security indicators across mobile browser apps based on W3C guidelines for web user interface. They conclude that mobile web browsers implement only a subset of desktop browsers' security indicators, leaving mobile users vulnerable to attacks that the indicators are designed to signal. While our study does not intend to evaluate the state of security indicators in Firefox OS, our observations could be used as a starting point for extrapolating the findings in [4] to Firefox OS. Akhawe et al. [2] quantify the effectiveness of desktop browser warnings by measuring their click-through rates. Their finding that a high proportion of users click through SSL warnings, although from a desktop browser user base, lends credence to the attack scenario based on a certificate override that we demonstrate.

2.5 Towards Semi-Automated Analysis

As we have seen in this chapter, the convergence of web technologies and smart devices leads to new security challenges. Security issues arise both at the application and systems level. To a great extent, Mozilla has relied on static analysis to facilitate application code review, while largely ignoring audit of systems code and the interaction between systems code and applications. Indeed, automated analysis of complex interactions between applications and the underlying system is a challenging task.

This chapter shows how a penetration testing approach, popular with security practitioners today, may be useful to audit such complex interactions. Nonetheless, this approach has an inherent drawback: Entirely manual security audit does not scale. In the following chapters, we set out to understand how vulnerability analysis tools may be designed so that manual labor involved in the task of vulnerability assessment is reduced. To this end, we envision an analysis framework that assists system implementors in specific tasks such as vulnerability identification, regression analysis, and fuzz testing.

3

Vulnerability Diagnosis Using Staged Program Analysis

This is the whole "C" attitude toward life, right? Life fast, die young, right?

– James Mickens, *MIT 6.858 Computer Systems Security*

In the previous chapter, we performed a security audit of the Firefox OS using a combination of manual code review and penetration testing. Although this work led to the discovery of two important security vulnerabilities, the adopted approach has two specific limitations. First, because it is labor intensive manual source code audits do not scale to large codebases. Second, penetration testing treats the program under test as a black-box due to which it may not discover vulnerabilities in deep program paths. Indeed, since no individual approach is capable of identifying all classes of vulnerabilities, diversifying vulnerability detection approaches is necessary. Prior studies [132, 7] have not only corroborated this observation in controlled environments but also concluded that more scalable vulnerability detection approaches such as static program analysis have the potential to boost programmer productivity.

In this chapter, we begin to conceptualize a static program analysis framework for vulnerability assessment. In particular, we describe the first of three methods called staged program analysis. Staged program analysis is aimed at diagnosing vulnerabilities that are spread across source files. It proceeds in two steps. For a structural program weakness (vulnerability class) known a priori, we first obtain vulnerability summaries—descriptions of potential vulnerabilities—at the function call level. Next, we evaluate these summaries at the whole program level to check if the vulnerabilities can indeed manifest in the scope of the whole program. Our staged approach keeps our analysis tractable to even large programs such as web browsers and helps diagnose vulnerabilities due to faulty interactions between software components.

The rest of this chapter is organized as follows. We continue with a brief discussion of the task of diagnosing distributed vulnerabilities in Section 3.1. We then present two primitives that will help us in this task, namely source (see Section 3.2.2) and whole program analyzers (see Section 3.2.3). We empirically evaluate our method in Section 3.3, and conclude this chapter with a discussion of related work in Section 3.4.

3.1 Task: Diagnosing Distributed Vulnerabilities

Today's software is complex and has many moving parts. It is natural that software development follows a divide-and-conquer strategy in which a high-level requirement is broken down into smaller development tasks. Programmers have a deep understanding of the components they create but only a shallow understanding of other components in the system. This may give rise to incorrect assumptions. For example, while one group of programmers may assume that the data provided to their component is well formed, another group may assume that sanity checks are carried out by the said component. We illustrate this problem using an otherwise straightforward memory corruption vulnerability in the Chromium codebase.

The code snippet in Listing 3.1 shows the implementation of a `PageIndicator` object that is used by several other components in the Chromium codebase. One of the object members called `fade_out_timer_id_` is defined lazily in the `ResetFadeOutTimer` method (line 11) such that invoking the `OnTimerFired` method reads uninitialized memory (line 18), potentially diverting program execution in an unintended way. This vulnerability was detected by the Google security team using dynamic program analysis tools [23].

Statically analyzing such defects calls for an approach that can analyze inter-component interactions at compile time with a low run time overhead. End-to-end analysis of entire codebases is ruled out because codebases such as Chromium have over 14 million source lines of code (SLoC). Our task is to create an analyzer that can not only scale up to large codebases but also provide sufficient diagnostics for the developer to fix those defects that manifest during inter-component interactions. To address these challenges, our method uses ideas from semantic code analysis and staged computation.

3.2 Staged Program Analysis

Although program analysis has long been used to flag software defects, their application to vulnerability assessment is recent. One of the biggest differences between

```
// Constructor definition of the PageIndicator object
// Object does not initialize member fade_out_timer_id_
PageIndicator::PageIndicator()
         : current_page_(0),
           splash_timeout_(kPageIndicatorSplashTimeoutMs),
           always_visible_(false) {}

// Method defines fade_out_timer_id_
void PageIndicator::ResetFadeOutTimer()
{
  fade_out_timer_id_ = owner()->ScheduleTimer(id(), splash_timeout_);
}

// Method assumes fade_out_timer_id_ initialized
void PageIndicator::OnTimerFired(uint32 timer_id)
{
  FadingControl::OnTimerFired(timer_id);
  if (timer_id == fade_out_timer_id_) {
    Fade(false, fade_timeout_);
  }
}
```

Listing 3.1: Security bug in Chromium version 38 in which reading uninitialized memory contents may lead to unintended control flow.

bug detection and vulnerability detection is that the latter depends a lot more on program run time information than the former. For example, memory corruption vulnerabilities—a large class of C/C++ vulnerabilities—exploit unintended reads from and writes to program memory at run time. Therefore, one of the biggest challenges in designing a *static* vulnerability analysis tool is to anticipate such dynamic behavior and generate informative bug reports when a violation is feasible.

Since our analyzer is invoked at program compile time, it does not have access to run time information. However, program semantics may be inferred via source code analysis. For example, if a source file declares a variable but fails to initialize it, it may be inferred that an uninitialized read is possible at run time. However, whether the uninitialized read is *feasible* is still not known. To this end, whole program analysis may be used. Fortunately, whole program analysis of specific feasibility queries is tractable because analysis is carried out on specific program variables. Therefore, a two-staged analysis in which a feasibility analysis examines specific instances of potential program defects is well-suited for the diagnosis of distributed vulnerabilities.

We design a staged program analyzer called Mélange. Figure 3.1 provides an overview of our approach. Mélange comprises four high-level components: the build interceptor, the LLVM builder, the source analyzer, and the Whole-Program (WP) analyzer. We summarize the role of each component in analyzing a program. Subsequently, we describe them in greater detail.

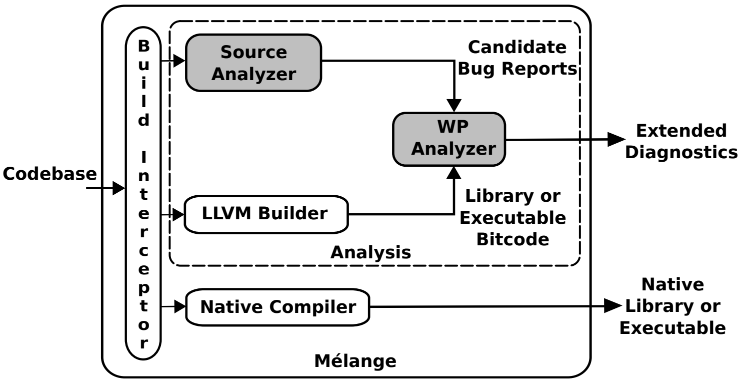


Fig. 3.1.: Overview of our staged program analyzer called Mélange.

1. *Build Interceptor.* The build interceptor is a program that interposes between the build program (e.g., *GNU-Make*) and the compilation system (e.g., Clang/LLVM). In Mélange, the build interceptor is responsible for *correctly* and *independently* invoking the program builders and the source analyzer. (§3.2.1)

2. *LLVM Builder.* The LLVM builder is a utility program that assists in generating LLVM Bitcode for `C`, `C++`, and `Objective-C` programs. It mirrors steps taken during native compilation onto LLVM Bitcode generation. (§3.2.1)

3. *Source Analyzer.* The source analyzer executes domain-specific checks on a source file and outputs candidate bug reports that diagnose a potential security bug. The source analyzer is invoked during the first stage of Mélange's analysis. We have implemented the source analyzer as a library of checkers that plug into a patched version of Clang SA. (§3.2.2)

4. *Whole-Program Analyzer.* The WP analyzer examines candidate bug reports (from Step 3), and either provides extended diagnostics for the report or classifies it as a false positive. The developer is shown only those reports that have extended diagnostics i.e., those not classified as a false positive by the WP analyzer. We have implemented the WP analyzer in multiple LLVM passes. (§3.2.3)

3.2.1 Analysis Utilities

Ease-of-deployment is one of the design goals of Mélange. We want software developers to use our analysis framework in their build environments seamlessly. The build interceptor and the LLVM builder are *analysis utilities* that help us achieve this goal. The build interceptor and the LLVM builder facilitate transparent analysis of codebases by *plugging in* Mélange's analyses to an existing build system. We describe them briefly in the following paragraphs.

Build Interceptor Our approach to transparently analyze large software projects hinges on triggering analysis via the build command. We use an existing build interceptor, scan-build [131], from the Clang project. scan-build is a command-line utility that intercepts build commands and invokes the source analyzer in tandem with the compiler. Since Mélange's WP analysis is targeted at program (LLVM) Bitcode, we instrument scan-build to not only invoke the source analyzer, but also the LLVM builder.

LLVM Builder Generating LLVM Bitcode for program libraries and executables without modifying source code and/or build configuration is a daunting task. Fortunately, the Whole-program LLVM (WLLVM) [157], an existing open-source LLVM builder, solves this problem. WLLVM is a python-based utility that leverages a compiler for generating whole-program or whole-library LLVM Bitcode. It can be used as a drop-in replacement for a compiler i.e., pointing the builder (e.g., *GNU-Make*) to WLLVM is sufficient.

3.2.2 Source Analyzer

The primary challenge of the source analyzer is to extract vulnerability artifacts from source code for subsequent investigation. The analyzer may be described as a system that takes a checking policy and extracts artifacts that conform to this policy. The concept of meta-compilation [6] offers a suitable solution for this task. A meta-compilation compilation system comprises a compiler and a set of checkers written as plugins to the compiler. While the compiler is responsible for performing code analysis and optimization, checkers are responsible for encoding a policy to be checked and emitting bug reports when this policy is violated. In the following, we describe the local analyzer.

The source analyzer flags potential bugs in source code. We build a novel *event collection* system that helps detect both taint-style vulnerabilities as well as semantic defects. Our event collection system is implemented as a system of taints on C and C++ language constructs (Declarations). We call the underlying mechanism Declaration Tainting because taints in the proposed event collection system are associated with AST Declaration identifiers of C and C++ objects. Since declaration tainting is applied on AST constructs, it can be carried out in situations where local symbolic execution is not possible.

We write checkers to flag defects. Checkers have been developed as *clients* of the proposed event collection system. The division of labor between checkers and the event collection system mirrors the Meta-level Compilation concept: Checkers encode the policy for flagging defects, while the event collection system maintains the state required to perform checks. We have prototyped this system for flagging garbage (uninitialized) reads[1] of C++ objects, incorrect type casts in PHP interpreter codebase, and other Common Weakness Enumerations (see §4.3).

We demonstrate the utility of the proposed system by using the code snippet shown in Listing 3.2 as a running example. Our aim is to detect uninitialized reads of class members in the example. The listing encompasses two source files, `foo.cpp` and `main.cpp`, and a header file `foo.h`. We maintain two sets in the event collection system: the `Def` set containing declaration identifiers for class members that have at least one definition, and the `UseWithoutDef` set containing identifiers for class members that are used (at least once) without a preceding definition. We maintain an instance of both sets for each function that we analyze in a translation unit i.e., for function F, Δ_F denotes the analysis summary of F that contains both sets. The checker decides how the event collection sets are populated. The logic for populating the `Def` and `UseWithoutDef` sets is simple. If a program statement in a given function defines a class member for the very first time, we add the class member identifier to the `Def` set of that function's analysis summary. If a program statement in a given function uses a class member that is absent from the `Def` set, we add the class member identifier to the `UseWithoutDef` set of that function's analysis summary.

In Listing 3.2, when function `foo::isZero` in file `foo.cpp` is being analyzed, the checker adds class member `foo::x` to the `UseWithoutDef` set of $\Delta_{foo::isZero}$ after analyzing the branch condition on Line 13. This is because the checker has not encountered a definition for `foo::x` in the present analysis context. Subsequently, analysis of the constructor function `foo::foo` does not yield any additions to either the `Def` or `UseWithoutDef` sets. So $\Delta_{foo::foo}$ is empty. Finally, the checker compares set memberships across analysis contexts. Since `foo::x` is marked as a use without a valid definition in $\Delta_{foo::isZero}$ and `foo::x` is not a member of the `Def` set in the constructor function's analysis summary ($\Delta_{foo::foo}$), the checker classifies the use of Line 13 as a candidate bug. The checker encodes the proof for the bug in the candidate bug report. Listing 3.3 shows how candidate bug reports are encoded. The

---

[1] The algorithm for flagging garbage reads is based on a variation of gen-kill sets [81].

```
1  // foo.h
2  class foo {
3  public:
4          int x;
5          foo() {}
6          bool isZero();
7  };
8
9  // foo.cpp
10 #include "foo.h"
11
12 bool foo::isZero() {
13   if (!x)
14     return true;
15 }
16
17 // main.cpp
18 #include "foo.h"
19
20 int main() {
21         foo f;
22         if (f.isZero())
23           return 0;
24         return 1;
25 }
```

Listing 3.2: Running example–The `foo` object does not initialize its class member `foo::x`. The call to `isZero` on Line 22 leads to a garbage read on Line 13.

bug report encodes the location and analysis stack corresponding to the potential garbage (uninitialized) read.

The proposed event collection approach has several benefits. First, by retrofitting simple declaration-based object tainting into Clang SA, we enable checkers to perform analysis based on the proposed taint abstraction. Due to its general-purpose nature, the taint abstraction is useful for discovering other defect types such as null pointer dereferences. Second, the tainting APIs we expose are opt-in. They may be used by existing and/or new checkers. Third, our additions leverage high-precision analysis infrastructure already available in Clang SA. We have implemented the event collection system as a patch to the mainline version of Clang Static Analyzer. In the next section, we describe how candidate bug reports are analyzed by our whole-program analyzer.

3.2.3 Whole-Program Analyzer

Whole-program analysis is demand-driven. Only candidate bug reports are analyzed. The analysis target is an LLVM Bitcode file of a library or executable. There are two aspects to WP analysis: Parsing of candidate bug reports to construct a query, and the analysis itself. We have written a simple python-based parser to parse candidate

```
// Source-level bug report
// report-e6ed9c.html
...
Local Path to Bug: foo::x->_ZN3foo6isZeroEv

Annotated Source Code
foo.cpp:4:6: warning: Potentially uninitialized object field
 if (!x)
    ^
1 warning generated.

// Whole-program bug report
---------- report-e6ed9c.html ---------
[+] Parsing bug report report-e6ed9c.html
[+] Writing queries into LLVM pass header file
[+] Recompiling LLVM pass
[+] Running LLVM BugReportAnalyzer pass against main
----------------------------------------
Candidate callchain is:

foo::isZero()
main
------------------------
```

Listing 3.3: Candidate bug report (top) and whole-program bug report (bottom) for garbage read in the running example shown in Listing 3.2.

bug reports and construct queries. The analysis itself is implemented as a set of LLVM passes. The bug report parser encodes queries as preprocessor directives in a pass header file. A driver script is used to recompile, and run the pass against all candidate bug reports.

Our whole-program analysis routine is composed of a `CallGraph` analysis pass. We leverage an existing LLVM pass called the `Basic CallGraph` pass to build a whole-program call graph. Since the basic pass misses control flow at indirect call sites, we have implemented additional analyses to improve upon the precision of the basic callgraph. Foremost among our analyses is Class Hierarchy Analysis (CHA) [43]. CHA enables us to devirtualize those dynamically dispatched call sites where we are sure no delegation is possible. Unfortunately, CHA can only be undertaken in scenarios where no new class hierarchies are introduced. In scenarios where CHA is not applicable, we examine call instructions to resolve as many forms of indirect call sites as possible. Our prototype resolves aliases of global functions, function casts etc.

Once program call graph has been obtained, we perform a vulnerability-specific WP analysis. For instance, to validate garbage reads, the pass inspects loads and store to the buggy program variable or object. In our running example (Listing 3.2), loads and stores to the `foo::x` class member indicated in candidate bug report (Listing 3.3) are tracked by the WP garbage read pass. To this end, the program call graph is traversed to check if a load of `foo::x` does not have a matching store. If all loads have a matching store, the candidate bug report is classified as a false positive. Otherwise, program call-chains in which a load from `foo::x` does not have a matching store are displayed to the analyst in the whole-program bug report (Listing 3.3).

3.3 Evaluation

We have evaluated Mélange against both static analysis benchmarks and real-world code. To gauge Mélange's utility, we have also tested it against known defects and vulnerabilities. Our evaluation seeks to answer the following questions:

- What is the effort required to use Mélange in an existing build system? (§3.3.1)
- How does Mélange perform against static analysis benchmarks? (§3.3.2)
- How does Mélange fare against known security vulnerabilities? (§3.3.3)
- What is the analysis run-time and effectiveness of Mélange against large well-tested codebases? (§3.3.4)

3.3.1 Deployability

Ease-of-deployment is one of the design goals of Mélange. Build interposition allows us to analyze codebases as is, without modifying build configuration and/or source code. We have deployed Mélange in an Amazon `compute` instance where codebases with different build systems have been analyzed (see §4.3.2). Another benefit of build system integration is incremental analysis. Only the very first build of a codebase incurs the cost of end-to-end analysis; subsequent analyses are incremental. While incremental analysis can be used in conjunction with daily builds, full analysis can be coupled with nightly builds and initiated on virtual machine clusters.

3.3.2 NIST Benchmarks

We used static analysis benchmarks released under NIST's SAMATE project [113] for benchmarking Mélange's detection rates. In particular, the Juliet C/C++ test suite (version 1.2) [114] was used to measure true and false positive detection rates for defects spread across multiple categories. The Juliet suite comprises test sets for

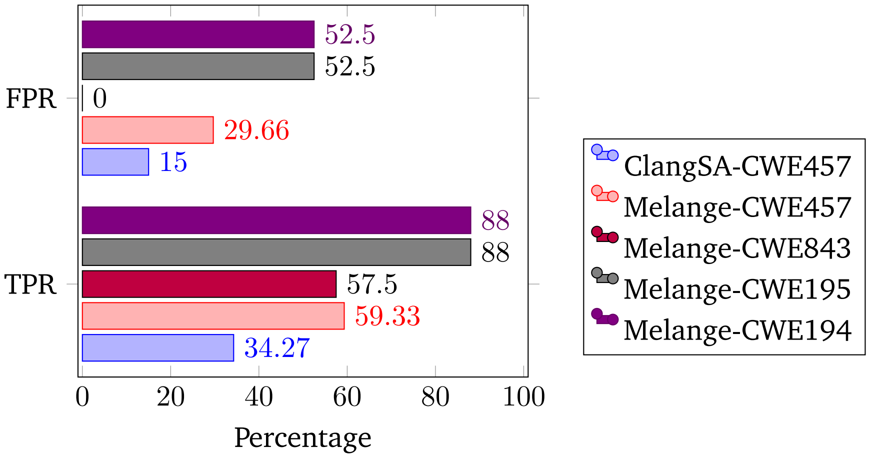


Fig. 3.2.: Juliet test suite: True Positive Rate (TPR) and False Positive Rate (FPR) for Mélange, and Clang Static Analyzer. Clang SA supports CWE457 only.

multiple defect types. Each test set contains test cases for a specific Common Weakness Enumeration (CWE) [149]. The CWE system assigns identifiers for common classes of software weaknesses that are known to lead to exploitable vulnerabilities. We implemented Mélange checkers and passes for the following CWE categories: CWE457 (Garbage or uninitialized read), CWE843 (Type confusion), CWE194 (Unexpected Sign Extension), and CWE195 (Signed to Unsigned Conversion Error). With the exception of CWE457, the listed CWEs have received scant attention from static analysis tools. For instance, type confusion (CWE843) is an emerging attack vector [86] for exploiting popular applications.

Figure 3.2 summarizes the True/False Positive Rates (TPRs/FPRs) for Clang SA and Mélange for the chosen CWE benchmarks. Currently, Clang SA only supports CWE457. Comparing reports from Clang SA and Mélange for the CWE457 test set, we find that the former errs on the side of precision (fewer false positives), while the latter errs on the side of caution (fewer false negatives). For the chosen CWE benchmarks, Mélange attains a true-positive rate between 57–88 %, and thus, it is capable of spotting over half of the bugs in the test suite.

Mélange's staggered analysis approach allows it to present both source file wide and program wide diagnostics (see Figure 3.3). In contrast, Clang SA's diagnostics are restricted to a single source file. Often, the call stack information presented in Mélange's extended diagnostics has speeded up manual validation of bug reports.

3.3.3 Detection of Known Vulnerabilities

We tested five known type-confusion vulnerabilities in the PHP interpreter with Mélange. All of the tested flaws are taint-style vulnerabilities: An attacker-controlled input is passed to a security-sensitive function call that wrongly interprets the input's type. Ultimately all these vulnerabilities result in invalid memory accesses that can be leveraged by an attacker for arbitrary code execution or information disclosure. We wrote a checker for detecting multiple instances of this vulnerability type in the PHP interpreter codebase. For patched vulnerabilities, testing was carried out on unpatched versions of the codebase. Mélange successfully flagged all known vulnerabilities. The first five entries of Table 3.1 summarize Mélange's findings. Three of the five vulnerabilities have been assigned Common Vulnerabilities and Exposures (CVE) identifiers by the MITRE Corporation. Reporters of `CVE-2014-3515`, `CVE-2015-4147`, and PHP report ID `73245` have received bug bounties totaling $5500 by the Internet Bug Bounty Panel [119].

In addition, we ran our checker against a recent PHP release candidate (PHP 7.0 RC7) released on 12th November, 2015. Thus far, Mélange has drawn attention to PHP sub-systems where a similar vulnerability may exist. While we haven't been able to verify if these are exploitable, this exercise demonstrates Mélange's utility in bringing attention to multiple instances of a software flaw in a large codebase that is under active development.

3.3.4 Case Studies

To further investigate the practical utility of Mélange, we conducted case studies with three popular open-source projects, namely, Chromium, Firefox, and MySQL. We focused on detecting garbage reads only. In the following paragraphs, we present results from our case studies emphasizing analysis effectiveness, and analysis runtime.

Software Versions: Evaluation was carried out for Chromium version 38 (dated August 2014), for Firefox revision 244208 (May 2015), and for MySQL version 5.7.7 (April 2015).

Evaluation Setup: Analysis was performed in an Amazon compute instance running Ubuntu 14.04 and provisioned with 36 virtual (Intel Xeon E5-2666 v3) CPUs clocked at 2.6 GHz, 60 GB of RAM, and 100 GB of SSD-based storage.

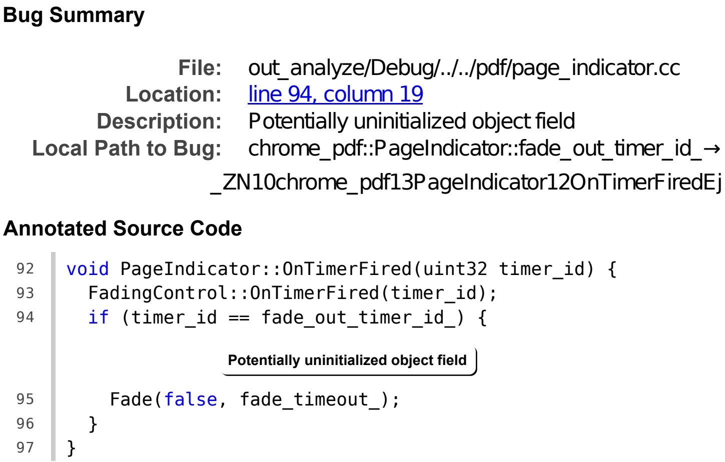
Codebase	CVE ID (Rating)	Bug ID	Vulnerability	Known/New
PHP	CVE-2015-4147	69085 [121]	Type-confusion	Known
PHP	CVE-2015-4148	69085 [121]	Type-confusion	Known
PHP	CVE-2014-3515	67492 [120]	Type-confusion	Known
PHP	Unassigned	73245 [127]	Type-confusion	Known
PHP	Unassigned	69152 [122]	Type-confusion	Known
Chromium	(Medium-Severity)	411177 [24]	Garbage read	Known
Chromium	None	436035 [25]	Garbage read	Known
Firefox	None	1168091 [19]	Garbage read	New

Tab. 3.1.: Detection summary of Mélange against production codebases. Mélange has confirmed known vulnerabilities and flagged a new defect in Firefox. Listed Chromium and Firefox bugs are not known to be exploitable. Chromium bug 411177 is classified as a Medium-Severity bug in Google's internal bug tracker.

Effectiveness

True Positives Our prototype flagged 3 confirmed defects in Chromium, and Firefox, including a new defect in the latter (see bottom three entries of Table 3.1). Defects found by our prototype in MySQL codebase have been reported upstream and are being triaged. Figure 3.3 shows Mélange's bug report for a garbage read in the pdf library shipped with Chromium v38. The source-level bug report (Figure 3.3a) shows the line of code that was buggy. WP analyzer's bug report (Figure 3.3b) shows candidate call chains in the libpdf library in which the uninitialized read may manifest.

We have manually validated the veracity of all bug reports generated by Mélange through source code audits. For each bug report, we verified if the data-flow and control-flow information conveyed in the report tallied with program semantics. We classified only those defects that passed our audit as true positives. Additionally, for the Chromium true positives, we matched Mélange's findings with reports [24, 25] generated by MemorySanitizer [148], a dynamic program analysis tool from Google. The new defect discovered in Firefox was reported upstream [19]. Our evaluation demonstrates that Mélange can complement dynamic program analysis tools in use today.



Bug Summary

File: out_analyze/Debug/../../pdf/page_indicator.cc
Location: line 94, column 19
Description: Potentially uninitialized object field
Local Path to Bug: chrome_pdf::PageIndicator::fade_out_timer_id_→ _ZN10chrome_pdf13PageIndicator12OnTimerFiredEj

Annotated Source Code

```
92  void PageIndicator::OnTimerFired(uint32 timer_id) {
93    FadingControl::OnTimerFired(timer_id);
94    if (timer_id == fade_out_timer_id_) {

                     Potentially uninitialized object field

95      Fade(false, fade_timeout_);
96    }
97  }
```

(a) Source-level Bug Report

```
---------- page_indicator.cc.pass.html ----------
[+] Parsing bug report page_indicator.cc.pass.html
[+] Writing queries into LLVM pass header file
[+] Recompiling LLVM pass
[+] Selecting LLVM BC for analysis
[+] Target Found: libpdf.a
[+] Running LLVM BugReportAnalyzer pass
------------------------
Candidate callchain is:
chrome_pdf::PageIndicator::OnTimerFired(unsigned int)
chrome_pdf::Instance::OnControlTimerFired(int,
unsigned int const&, unsigned int)
```

(b) Whole-program Bug Report

Fig. 3.3.: Mélange bug report for Chromium bug 411177.

False Positives Broadly, we encounter two kinds of false positives; those that are due to imprecision in Mélange's data-flow analysis, and those due to imprecision in its control-flow analysis. In the following paragraphs, we describe one example of each kind of false positive.

Data-flow imprecision: Mélange's analyses for flagging garbage reads lack sophisticated alias analysis. For instance, initialization of C++ objects passed-by-reference is missed. Listing 3.4 shows a code snippet borrowed from the Firefox codebase that illustrates this category of false positives.

When `AltSvcMapping` object is constructed (see Line 2 of Listing 3.4), one of its class members `mHttps` is passed by reference to the callee function `SchemeIsHTTPS`. The callee function `SchemeIsHTTPS` initializes `mHttps` via its alias (`outIsHTTPS`). Mélange's garbage read checker misses the aliased store and incorrectly flags the use of class member `mHttps` on Line 8 as a candidate bug. Mélange's garbage read pass, on its part, tries to taint all functions that store to `mHttps`. Since the store to `mHttps` happens via an alias, the pass also misses the store and outputs a legitimate control-flow sequence in its WP bug report.

```
AltSvcMapping::AltSvcMapping(...) {
  if (NS_FAILED(SchemeIsHTTPS(originScheme, mHttps))) {
    ...
  }
}
void AltSvcMapping::GetConnectionInfo(...) {
  // ci is an object on the stack
  ci->SetInsecureScheme(!mHttps);
  ...
}
static nsresult SchemeIsHTTPS(const nsACString &originScheme, bool &
    outIsHTTPS)
{
  outIsHTTPS = originScheme.Equals(NS_LITERAL_CSTRING("https"));
  ...
}
```

Listing 3.4: Code snippet involving an aliased definition that caused a false positive in Mélange.

Control-flow imprecision: Mélange's WP analyzer misses control-flow information at indirect call sites e.g., virtual function invocations. Thus, class members that are initialized in a call sequence comprising an indirect function call are not registered by Mélange's garbage read pass. While resolving all indirect call sites in large programs is impossible, we employ best-effort devirtualization techniques such as Rapid Type Analysis [9] to improve Mélange's control-flow precision.

Codebase	SLoC (millions)	Compilation Time N_t	Analysis Overhead SA_x	WPA_x	TA_x	$WPAvg_t$
Chromium	14.7	18m20s	29.09	15.49	44.58	7.5s
Firefox	4.9	41m25s	3.38	39.31	42.69	13m35s
MySQL	1.2	8m15s	9.26	21.24	30.50	2m26s

Tab. 3.2.: Mélange: Analysis Summary for Large Open-source Projects. All terms except $WPAvg$ are normalized to native compilation time.

Codebase	Number of Bug Reports Source (q)	Whole-program
Chromium	2686	12
Firefox	587	16
MySQL	2494	32

Tab. 3.3.: Mélange: Number of bugs reported after source-level and whole-program analysis.

The final three columns of Table 3.2 present a summary of Mélange's findings for Chromium, Firefox, and MySQL projects. We find that Mélange's two-stage analysis pipeline is very effective at filtering through a handful of bug reports that merit attention. In particular, Mélange's WP analyses filter out 99.6%, 97.3%, and 98.7%

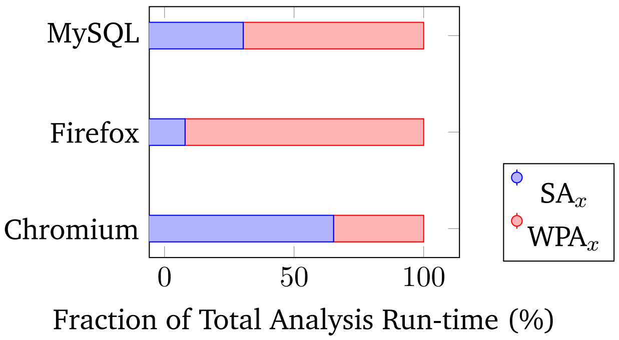


Fig. 3.4.: For each codebase, its source and whole-program analysis run-times are shown as fractions (in %) of Mélange's total analysis run-time.

source level bug reports in Chromium, Firefox, and MySQL respectively. Although Mélange's true positive rate is low in our case studies, the corner cases it has pointed out, notwithstanding the confirmed bugs it has flagged, is encouraging. Given that we evaluated Mélange against well-tested production code, the fact that it could point out three confirmed defects in the Chromium and Firefox codebases is a promising result. We plan to make our tool production-ready by incorporating insights gained from our case studies. Next, we discuss Mélange's analysis run-time.

Analysis Run-Time

We completed end-to-analysis of Chromium, Firefox, and MySQL codebases—all of which have millions of lines of code—in under 48 hours. Of these, MySQL, and Chromium were analyzed in a little over 4 hours, and 13 hours respectively. Table 3.2 summarizes Mélange's run-time for our case studies. We have presented the analysis run-time of a codebase relative (normalized) to its build time, N_t. For instance, a normalized analysis run-time of 30 for a codebase indicates that the time taken to analyze the codebase is 30**x** longer than its build time. All normalized run-times are denoted with the x subscript. Normalized source analysis time, WP analysis time, and total analysis time of Mélange are denoted as SA_x, WPA_x, and TA_x respectively. The term $WPAvg_t$ denotes the average time (not normalized) taken by Mélange's WP analyzer to analyze a single candidate bug report.

Figure 3.4 shows source and WP analysis run-times for a codebase as a fraction (in percentage terms) of Mélange's total analysis run-time. Owing to Chromium's modular build system, we could localize a source defect to a small-sized library. The average size of program analyzed for Chromium (1.8MB) was much lower compared to MySQL (150MB), and Firefox (1.1GB). As a consequence, the WP analysis run-times for Firefox, and MySQL are relatively high. While our foremost priority while prototyping Mélange has been functional effectiveness, our implementation leaves significant room for optimizations that will help bring down Mélange's end-to-end analysis run-time.

3.3.5 Limitations

Approach Limitations By design, Mélange requires two analysis procedures at different code abstractions for a given defect type. We depend on programmer-written analysis routines to scale out to multiple defect types. Two actualities lend credence to our approach: First, analysis infrastructure required to carry out extended analyses is already available and its use is well-documented. This has assisted us in prototyping Mélange for four different CWEs. Second, the complexity of analysis routines is many times lower than the program under analysis. Our analysis procedures span $2,598$ lines of code in total, while our largest analysis target (Chromium) has over 14 million lines of C++ code.

While Mélange provides precise diagnostics for security bugs it has discovered, manual validation of bug reports is still required. Given that software routinely undergoes manual review during development, our tool does not introduce an additional requirement. Rather, Mélange's diagnostics bring attention to problematic corner cases in source code. The manual validation process of Mélange's bug reports may be streamlined by subsuming our tool under existing software development processes (e.g., nightly builds, continuous integration).

Implementation Limitations Mélange's WP analysis is path and context insensitive. This makes Mélange's whole-program analyzer imprecise and prone to issuing false warnings. To counter imprecision, we can augment our WP analyzer with additional analyses. Specifically, more powerful alias analysis and aggressive devirtualization algorithms will help prune false positives further. One approach to counter existing imprecision is to employ a ranking mechanism for bug reports (e.g., Z-Ranking [83]).

3.4 Related Work

Lint was developed at Bell Labs in 1977, primarily to check "portability, style, and efficiency" of C programs [76]. Lint performed syntactic analysis, predominantly type-checking, of programs for flagging potential programming errors. In the past two decades, many commercial [33, 62, 80, 67], closed-source [26], free [64], and open source [90, 136, 148, 10, 66, 50, 156, 21, 8, 160] tools have been developed. Broadly, these tools are based on *Model Checking* [10, 66], *Theorem Proving* [64], *Static Program Analysis* [90, 67, 33, 62, 26, 160], *Dynamic Analysis* [108, 136, 148, 21], or hybrid systems such as [8]. In the following paragraphs, we comment on related work that is close in spirit to Mélange.

Program Instrumentation Traditionally, memory access bugs have been found by performing randomized program testing (e.g., fuzzing) of instrumented program binaries. The instrumentation takes care of tracking the state of program memory and adds run-time checks before memory accesses are made. Instrumentation is done either during run time (as in Valgrind [108]), or at compile time (as in AddressSanitizer [136]). Since compile-time instrumentation has a lower run time overhead, the idea has gained traction, and compile-time program instrumentation is now being used to find multiple kinds of memory access errors including reads from uninitialized memory [148]. As effective as tools such as Valgrind and ASan are, they fail to provide an assurance on the program as a whole. In part, this is because the testing methodology that underlies these tools is empirical: program is tested against concrete inputs that may or may not exercise all possible program paths (Code Coverage). The code coverage problem gets exacerbated when the program under test has millions of lines of code. To the best of our knowledge, we are unaware of published work that evaluates code coverage of fuzzer-based tools against large codebases. In contrast, a static analysis tool such as ours, performs more comprehensive code analysis, complementing software testing.

Symbolic Execution Symbolic execution has been used to find bugs in programs, or to generate test cases with improved code coverage. KLEE [21], Clang SA [90], and AEG [8] use different flavors of forward symbolic execution for their own end. In KLEE and AEG, FSE is performed at run-time. As the program (symbolically) executes, constraints on program paths (path predicates) are maintained. Satisfiability queries on path predicates are used to prune infeasible program paths. AEG proposes heuristics to reduce input space of symbolic inputs using a technique the authors call “preconditioned symbolic execution.” Unlike KLEE and AEG, symbolic execution in Clang SA is done statically. Moreover, the extent of symbolic execution in Clang SA is limited to a single source file, unlike KLEE and AEG that symbolically execute whole-programs. Anecdotal evidence suggests that KLEE and AEG don’t scale up to large programs like web browsers [65].

UC-KLEE [126] offers an alternate mode for symbolic execution which its authors describe as “under-constrained.” UC-KLEE, unlike KLEE, begins symbolical execution at procedure entry points. By skipping over constraints that would have been built up by a whole-program symbolic analyzer such as KLEE, UC-KLEE achieves greater scalability. Mélange shares the larger goal of reconciling analysis scalability and analysis effectiveness with UC-KLEE. However, Mélange focuses on finding defects in object-oriented code. Indirections that object-oriented code introduce, necessitates analysis whose outlook is global in nature.

Static Analysis Parfait [26] employs an analysis strategy that is similar in spirit to ours for finding buffer overflows in C programs. It employs multiple stages of analysis. In Parfait, each successive stage is more precise than the preceding stage. Parfait uses IR-only analysis in contrast to mixed-level (Source + IR) analysis that we undertake. We want source-level patterns that typically characterize bug classes (e.g., uninitialized read, use-after-free) to be driving analysis. This precludes IR-only analysis. Like Yamaguchi et al. [160], our goal is empower developers in finding multiple instances of a known defect. However, the approach we take is different. In [160], structural traits in a program’s AST representation are used to drive a Machine Learning (ML) phase. The ML phase *extrapolates* traits of known vulnerabilities in a codebase, obtaining matches that are similar in structure to the vulnerability. In contrast, Mélange employs semantic analysis to drive defect discovery, while leveraging an LLVM pass towards defect extrapolation. Our notion of defect extrapolation is different: For us, extrapolation entails obtaining whole-program diagnostics for a local defect.

CQUAL [50], and CQual++ [156], are flow-insensitive data-flow analysis frameworks for C and C++ languages respectively. Oink performs whole-program data-flow analysis on the back of Elsa, a C++ parser, and Cqual++. Data-flow analysis is based on type qualifiers. Our approach has two advantages over Cqual++. We use a production compiler for parsing C++ code that has a much better success rate at parsing advanced C++ code than a custom parser such as Elsa. Second, our source-level analysis is both flow and path sensitive while, in CQual++, it is not. Livshits et al. [89], propose a program analysis technique to detect security vulnerabilities in Java code. Their technique relies on context-sensitive points-to analysis of Java objects. Java bytecode is analyzed against a user-specified vulnerability specification written in a declarative language. The approach is similar in spirit to Fortify’s taint analyzer [143] that performs analysis of IR-level objective C code using user-written tainting rules. Both these works are geared towards IDE integration. In contrast, Mélange encodes a hard-coded policy in its source-level checker. However, our taint infrastructure can be leveraged to implement a system that accommodates user-specified taint policy as in [89, 143].

Finally, Clang Static Analyzer borrows ideas from several publications including (but not limited to) [128, 63]. Inter-procedural context-sensitive analysis in Clang SA is based on graph reachability that was proposed by Reps et al. [128]. Clang SA is similar in spirit to xgcc [63].

4

Vulnerability Diagnosis Using Static Input Format Inference

> *In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior.*
>
> – IETF RFC 791, *Internet Protocol Specification*

In the previous chapter, we demonstrated the utility of staged program analysis in diagnosing vulnerabilities that are caused by faulty interactions between program components. So far, we have taken a program-centric view of vulnerabilities. Program *input* has been of no consequence in our analyses. In practice, several applications such as network and file format parsers that parse highly structured input can benefit from input-centric analysis.

Parsing applications contain an input processing pipeline in which 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 for traditional program-centric analyzers. First, the highly structured nature of input leads to 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 analyzers. Second, the diversity of program input not only amplifies the number of control flows but also demands analysis in breadth. For example, the deep packet inspection library, nDPI, analyzes close to 200 different network protocols [107]. In the face of such diversity, generating inputs that efficiently test application logic is a hard problem.

In this chapter, we address these challenges by using static analysis to assist dynamic program testing tools such as fuzzers. Specifically, we propose a static analysis method that accepts source code as input, and infers the structure of program input processed by the source code. The inferred input structure is encoded as a dictionary and supplied to the fuzzer towards program testing. Our method for input inference is based on the insight that code patterns reflect the specification of inputs processed by a program. For example, we expect that a parsing program matches constant tokens in its input stream by encoding them as constant strings. To this end, we analyze code to infer the underlying data specification.

An important consequence of Rice's theorem [130] is that program analysis is incomplete in general. In the scope of this work, this means that the data format specification may not be completely inferred by a static analyzer. However, our evaluation shows that *even* incomplete descriptions of this specification can make program testing more effective in two ways: They can increase test coverage, find more vulnerabilities or do both. Overall, the presented method increases confidence in software that is going to be shipped to end-users, contributing to the aim of this dissertation.

The rest of this chapter is organized as follows. We proceed with a definition of the task of inferring the input specification in Section 4.1. We then present the design and implementation of a static analyzer geared towards extracting input fragments from source code in Section 4.2. We evaluate our method in controlled as well as uncontrolled environments. Our evaluation methodology and results are presented in Section 4.3. We conclude the chapter with a discussion of related work in Section 4.4.

4.1 Task: Input Inference

Parsing applications are an integral part of computer networks, being responsible for correctly decoding data exchanged over the network. These applications are a natural target for attackers because they are usually remotely accessible and afford a foothold into private infrastructure. Due to their sensitive nature, parsing applications must be conservative in what they accept failing safely when input is malformed. Instead, networking protocols have been historically designed to be liberal in what they accept drawing attackers' interest to parsing applications.

Testing the resilience of parsers to malformed input is a big challenge. Consider the code responsible for decoding Internet protocol (IP) packets in the Snort++ intrusion detection system shown in Listing 4.1. The `Ipv4Codec::decode` function accepts raw IPv4 packet data and a decoding configuration as input, decoding the packet into an internal data structure. The IPv4 decoder needs to perform tens of checks to ensure that IPv4 data is well formed. As illustrated in Listing 4.1 these checks are usually encoded as branch statements such as `if` statements. For example, IPv4 version check is encoded in an `if` statement on line 12, header length check on line

```
 1 bool Ipv4Codec::decode(const RawData& raw, CodecData& codec, DecodeData&
       snort)
 2 {
 3     ...
 4     if (raw.len < ip::IP4_HEADER_LEN)
 5     {
 6         if ((codec.codec_flags & CODEC_UNSURE_ENCAP) == 0)
 7       ...
 8     }
 9
10     if ( snort::SnortConfig::get_conf()->hit_ip_maxlayers(codec.ip_layer_cnt)
           )
11   ...
12     if (iph->ver() != 4)
13     {
14         if ((codec.codec_flags & CODEC_UNSURE_ENCAP) == 0)
15             ...
16     }
17
18     if (hlen < ip::IP4_HEADER_LEN)
19   ...
20     if (ip_len > raw.len)
21   ...
22     if (ip_len < hlen)
23   ...
24     if ( snort.ip_api.is_ip6() )
25   ...
26 }
```

Listing 4.1: Snort++ code snippet for decoding IP packets.

18, total payload length on line 20 etc. Since the number of program paths grows exponentially with the number of branch statements, software testing tools such as fuzzers find it difficult to test each and every program path. Moreover, since the input to parsers is highly structured, program paths are exercised only if program input contains specific tokens of interest.

This setting calls for a method that can augment input-centric testing with program information so that implementors can obtain a high assurance that parsers are fail safe. To this end, our method uses static program analysis to obtain input-centric information about the program in advance so that test effectiveness can be improved.

4.2 Data Format Dictionary Construction

It is evident from our previous discussion that awareness of the input format is crucial for improving test effectiveness. Next, we first briefly outline the problem scope with regard to protocol specification inference, then provide an overview of our approach, and finally describe our methodology.

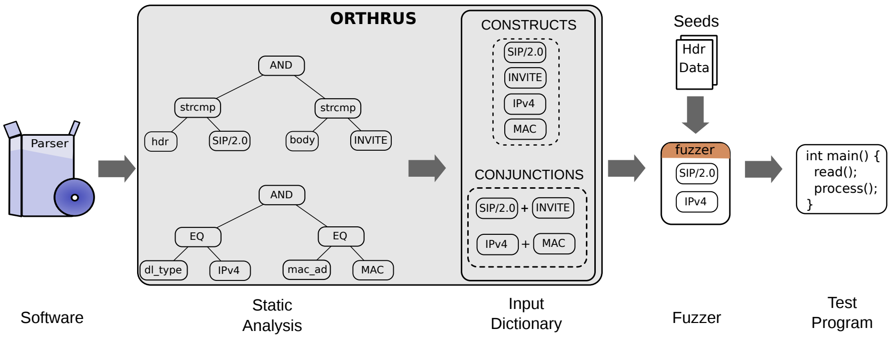


Fig. 4.1.: Work-flow for program analysis guided fuzzing.

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 [162]. However, creating input dictionaries has required 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 4.1 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 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 software instrumentation. 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.

4.2.1 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 [69] 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 [52]: 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 pre-determined 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, it is tractable and scales up to large programs. Furthermore, we can efficiently obtain backward program slices using analysis infrastructure that comes with modern compilers. The selection criteria for taint sinks influence analysis precision, and ultimately decide the quality of inferred input fragments. To this end, 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. Subsequently, we describe how we perform backward slicing with reference to the chosen sinks, using analysis queries.

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

1. Is a branch statement involving a program variable e.g., `switch...case`, `If` etc.
2. Is a well-known data sink e.g., POSIX functions `strcmp`, `memcmp` etc.
3. Contains a `const` qualified program variable such as
   `const char *sip = "SIP/2.0"`
4. Contains a literal string, character, or an integer such as
   `if(mpls == 0x8848)`

Although these heuristics are simple, they are effective, and have two useful properties. First, they narrow the search for protocol keywords (e.g., Heuristic 2, 3, 4), or potentially data-dependent control flow (Heuristic 1). Second, although our heuristics are not sound, they capture a wide array of code patterns that are commonly found in parsing applications. Thus, they constitute a good *default specification* that is broadly 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. To cater to such use cases, we permit the security analyst to specify a non-default set of taint sinks as an analysis parameter. Thus, our analysis framework facilitates automatic analysis of third-party software, while opportunistically benefiting from application-specific knowledge. 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). Equipped with the queries, for each source file, we obtain a list of input message constructs present in it using syntactic analysis (line 11), and conjunctions of constructs using semantic analysis (line 13). The constructs and conjunctions obtained from the source file is added to the dictionary data structure(line $14-15$) and the analysis continues on the next source file.

Algorithm 1 Pseudocode for generating an input dictionary.

```
function GENERATE-DICTIONARY($SourceCode$, $Builder$)
    dictionary = $\emptyset$
    constructs = $\emptyset$
    conjunctions = $\emptyset$
    ▷ Queries generated from internal database
    queries = $Q$
    for each $sourcefile$ in $SourceCode$ do
        ast = frontendParse($sourcefile$)
        cfg = semanticParse($ast$)
        ▷ Obtain constructs
        constructs = syntactic-analysis($ast$, $queries$)
        ▷ Obtain conjunctions of existing constructs
        conjunctions = semantic-analysis($cfg$, $constructs$)
        ▷ Update dictionary
        dictionary += constructs
        dictionary += conjunctions
    return dictionary

function SYNTACTIC-ANALYSIS($AST$, $Queries$)
    constructs = $\emptyset$
    for each $query$ in $Q$ do
        constructs += synQuery($AST$, $query$)
    return constructs

function SYNQUERY($AST$, $Query$)
    matches = $\emptyset$
    while T = traverseAST($AST$) do
        if $Query$ matches T then
            matches += (T.id, T.value)
    return matches

function SEMANTIC-ANALYSIS($CFG$, $Constructs$)
    conjunctions = $\emptyset$
    ▷ Obtain conjunctions in a given calling context
    conjunctions += Context-Sensitive-Analysis($CFG$, $Constructs$)
    ▷ Obtain productions in a given program path
    conjunctions += Path-Sensitive-Analysis($CFG$, $Constructs$)
    return conjunctions
```

Syntactic Queries: At the syntactic level, our analysis logic accepts functional queries and returns input message constructs (if any) that match the issued query. These queries are made against the program AST. In our framework, a functional query is a query that 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 elicits 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. Second, since large parsers use a recurring pattern of code to parse input messages of different formats, even simple queries are efficient at building a multi-protocol input dictionary.

Syntactic queries are efficient 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 [129] 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 4.2.

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.

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

```c
int parse(const char *token1, const char *token2) {
  if (token1 == "INVITE")
    if (strcmp(token2, "SIP/2.0"))
      do_something();
}
```

Listing 4.2: Sample parser code.

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($input$)
```

random mutations. Next, we document how our input dictionary can be integrated into a modern fuzzer.

4.2.2 Fuzzing with an Input Dictionary

In order to efficiently test applications with highly structured input, 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 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 be high. Our use of demand-driven queries, and analyses of varying precision ensures that we supply such a dictionary to the fuzzer.

Soundness and Precision: In summary, by providing message format specific insight to the fuzzer in advance, our approach should increase the test coverage. However, owing to the unsound nature of our chosen heuristics, it is possible that our approach fails to extract certain input fragments i.e., has false negatives. However, in scenarios where application-specific knowledge is inevitable, we permit the user to supply custom analysis parameters, such as, application-specific taint sinks and functional queries on them. This lowers the number of false negatives, at an acceptable cost to usability.

Our semantic analysis routines have a high degree of precision, owing to their context and path sensitivity. Thus, input conjunctions returned by our semantic queries have a low number of false positives.

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 [90] (semantic queries). To enable ease-of-use, we have packaged our framework as a test pipeline that is scripted in the Python language. Although we have described our proposal in the context of protocol parsers, our approach is generalizable to other classes of parsing applications such as file format parsers. At the moment, our prototype may be used to conduct grammar-based security assessments of applications written in the C/C++ programming language.

4.3 Evaluation

In this section, we present our evaluation of Orthrus in both controlled and uncontrolled environments. First, we assess the time Orthrus needs to uncover vulnerabil-

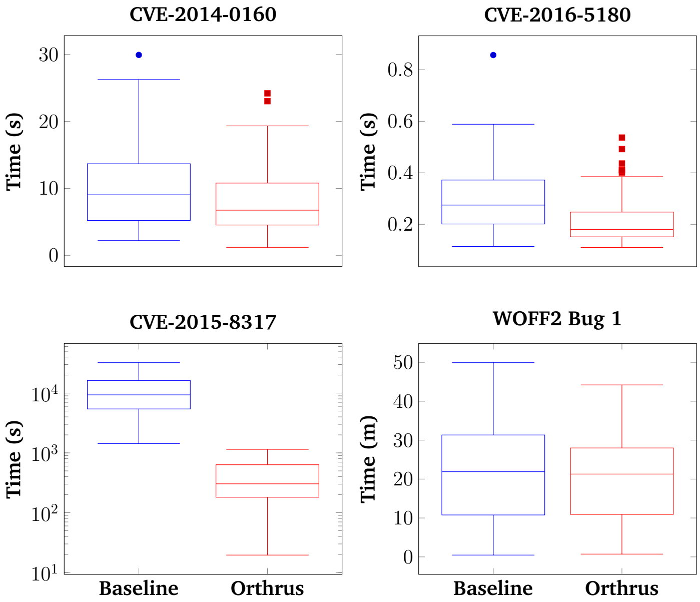


Fig. 4.2.: Comparison of time required to expose vulnerability using libFuzzer as the baseline.

ities in a set of fuzzer benchmarks (§4.3.1). Subsequently, we evaluate how well Orthrus performs in comparison to state-of-the-art fuzzers on real-world codebases (§4.3.2).

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.

4.3.1 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 [61]. 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 [91], 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 [96] (OpenSSL Heartbleed), CVE-2016-5180 [98] (buffer overflow in the c-ares dns library), CVE-2015-8317 [97] (buffer overflow in libxml2), and a security-critical bug in Google's WoFF2 font parser [29].

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 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 each of the two scenarios. Measurements for each test case were carried out in parallel, with one experiment being run 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 4.2 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) to the right. 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 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 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.

4.3.2 Case Study

To investigate the practical utility of Orthrus, we conducted a case study of two popular network applications, namely, nDPI, and tcpdump. For each application, we conducted multivariate testing using baseline fuzzers such as afl and aflfast [16] with and without Orthrus generated dictionary.

The chosen applications were also fuzzed using the Peach fuzzer [117], 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 prevented 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 [100] 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.

To minimize the impact of seed quality on fuzzer outcome, two elementary seeds (bare-bones IPv4, and IPv6 packets) were used in all our evaluations. It is possible that a diverse set of seeds improves the effectiveness of fuzzing. Hence, we have carefully analyzed the additional coverage achieved solely through the use of input

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

Tab. 4.1.: Test coverage achieved by different fuzzing configurations.

fragments in the supplied dictionary to ensure that the presented increments can be attributed to our method. In addition, we manually triage all the vulnerabilities that were exclusively found using our approach to ensure causality i.e., that they are identified due to the use of specific tokens in the supplied dictionary. Tests involving the fuzzers afl and aflfast were conducted in a multi-core setting.

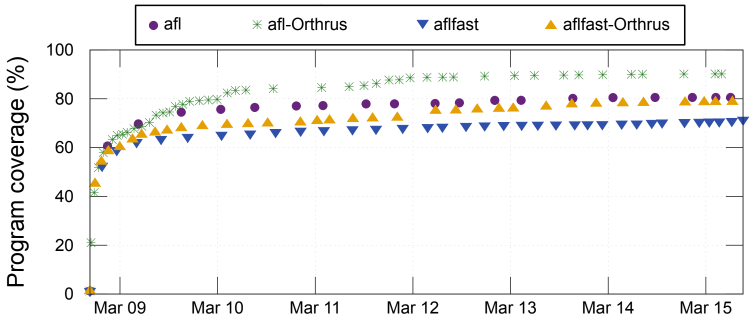
Fuzzing Duration: Dictionary based mutations get a fraction of the total fuzz time of a fuzzer. Thus, to fully evaluate our approach, we ran the fuzzer configurations (except Peach) until each unique program input synthesized by the fuzzer was mutated with the supplied dictionary constructs at least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

Utilities: CERT's `exploitable` [49] utility was used for crash deduplication. We used AddressSanitizer [136] as a debugging aid; this expedited the bug reporting process.

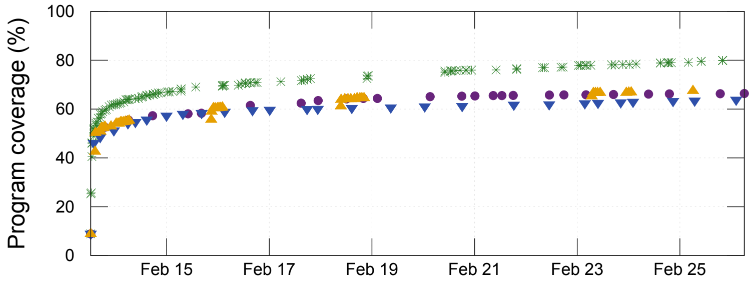
Evaluated Software: We evaluated nDPI revision f51fef6 (November 2016), and tcpdump trunk (March 2017).

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 4.1 shows the test coverage achieved by different fuzzer combinations for tcpdump, and nDPI, while Figure 4.3 visualizes code coverage over time. Coverage measurements have been performed whenever there is a change in coverage, as reported by the



(a) tcpdump



(b) nDPI

Fig. 4.3.: Test coverage as a function of time for tcpdump 4.3a, and nDPI 4.3b, for different fuzzing configurations. Program coverage measurements were made only when there was a change in its magnitude.

fuzzer. Due to the relatively short running duration of the Peach fuzzer, we have excluded its coverage visualization.

As shown in Figure 4.3, 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 4.1. 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

Tab. 4.2.: 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

aggressive, resulting in too few inputs in the fuzzer queue undergoing dictionary mutations.

Vulnerabilities Exposed

Table 4.2 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 hypothesize that the fuzz schedule alterations carried out in aflfast [16] influence the scheduling of dictionary-mutations, resulting in the observed drop.

Table 4.3 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 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.

Tab. 4.3.: 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

4.4 Related Work

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

Inferring Protocol Specification: There are two problems underlying protocol specification inference: Inferring the (i) protocol message format; and (ii) protocol state machine. Prior work, with the exception of Prospex [30] 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 [35]. 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 [20], Tupni [36], Autoformat [88], Prospex [30], and the tool by Wondracek et al. [158] 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 the protocol inference is as simple as compiling the underlying source code. Second, prior works with the exception of Prospex, have not specifically evaluated the impact of their inference on the effectiveness of fuzz testing. Although Comparetti et al. [30] 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 works, 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. [57] 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 [68] 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. [84], and automatic inference of search patterns for discovering taint-style vulnerabilities from source code by Yamaguchi et al [159]. 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.

Vulnerability Diagnosis Using Static Template Matching

5

> *If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.*
>
> *– The Duck Test*

In the previous chapter, we saw that the effectiveness of fuzz testing can be improved by an awareness of the program input format. To this end, we were able to use static analysis to extract input fragments from source code. Although, static analysis augmented fuzzing helped us uncover tens of zero-day vulnerabilities, we witnessed that test coverage fell short of 100%. A natural question that follows from this observation is whether *all* vulnerabilities have been uncovered or not? Although it is impossible to be certain, we can always hope to improve our confidence in the program by more detailed analysis. The overall aim of this chapter is to increase the confidence in the program by diagnosing vulnerabilities missed by fuzzers.

Although fuzz testing can only prove presence and not absence of vulnerabilities, it provides valuable feedback for taking vulnerability diagnosis further. As we have seen in the previous chapter, a program crash uncovered using fuzz testing can help locate memory corruption vulnerabilities that lay dormant in the program. Our intuition is that we can apply template matching to locate other instances of a vulnerability flagged by a fuzzer. This requires us to understand code properties of a vulnerability and leverage them for furthering our vulnerability search. We approach this problem using concepts from automatic fault localization and template matching. The method presented in this chapter may not only be used to explore known vulnerability patterns but to also perform regression analysis: eliminating the recurrence of known vulnerabilities across a codebase.

The rest of the chapter is structured as follows. We begin with a brief discussion of the task of static exploration of fuzzer crashes in Section 5.1. We then present the design and implementation of our method in Section 5.2. We evaluate our method using a case study of Open vSwitch (OvS), an open-source virtual switch

```
#include <string.h>
#include <crypt.h>
#include <stdlib.h>
#include <unistd.h>
#define CUSTOM() abort()
void fuzzable(const char *input) {
    // Fuzzer finds this bug
    if (!strcmp(input, "doom"))
        abort();
}

void cov_bottleneck(const char *input) {
    char *hash = crypt(input, "salt");

    // Fuzzer is unlikely to find this bug
    if (!strcmp(hash, "hash_val"))
       CUSTOM(); // grep misses this
}

// Fuzzer test harness
// INPUT: stdin
int main() {
    char buf[256];
    memset(buf, 0, 256);
    read(0, buf, 255);
    fuzzable(buf);
    cov_bottleneck(buf);
    return 0;
}
```

Listing 5.1: A representative fuzzer test harness in which two synthetic denial of service vulnerabilities have been introduced by calling the `abort()` function.

implementation used in data centers in Section 5.3. Finally, we conclude the chapter with a discussion of related work in Section 5.4.

5.1 Task: Static Exploration of Fuzzer Crashes

Although fuzz testing is a simple and effective technique for vulnerability discovery, it can not be used to test all program paths. This is a practical difficulty encountered by security practitioners on a regular basis. We illustrate this problem using a synthetic code example that is shown in Listing 5.1. This program accepts potentially attacker controlled input, invoking two APIs, namely `fuzzable` and `cov_bottleneck` on this input. Note that both these APIs may call the `CUSTOM` function which is a short hand for the `abort` C library call that abruptly terminates a program leading to a denial of service.

In particular, the program aborts while parsing (i) the input string literal `doom`; (ii) the input whose cryptographic hash equals the string `hash_val`. We assume that the fuzzer is able to quickly find the crash due to the string literal comparison (`doom`), but is unlikely to generate the input that satisfies the cryptographic operation. This is a reasonable assumption since a fuzzer is very unlikely to synthesize a hash collision simply by performing random input mutations. This means that the second vulnerability, although identical to the first, will probably never be found by fuzz testing alone. Therefore, even after fuzz testing has been conducted and the first vulnerability has been uncovered, the security practitioner is left wondering if the same vulnerability may manifest in untested program paths. This setting calls for a method that can accept a known program crash as input and perform a search for recurring instances of the underlying vulnerability. We refer to this task as vulnerability exploration.

Static exploration of fuzzer crashes poses three challenges. Exploring a vulnerability pattern requires us to know what the pattern *is* to begin with. This requires us to automatically formulate a vulnerability pattern from a program crash. Next, we need to develop algorithms and data structures to facilitate vulnerability search based on this pattern. Finally, the output of the vulnerability search must facilitate easy diagnosis and fixing of recurring vulnerabilities. We address these challenges by using concepts from spectrum based fault localization and program analysis.

5.2 Static Exploration

Contemporary fuzzers and dynamic memory analysis tools have greatly advanced vulnerability detection and re-mediation, owing to their ease-of-use and public availability. Since fuzzers and memory analyzers are invoked at runtime, they require a test harness that accepts user input (usually read from a file or standard input), and invokes program APIs against this input. Therefore, the effectiveness of fuzzing and dynamic memory analysis depends on the availability of test cases that exercise a wide array of program APIs.

In practice, code bases contain test harnesses for only a limited number of program APIs. This means that, even if fuzzing were to achieve 100% test coverage (either line or edge coverage) for the set of *existing* test harnesses, it does not lead to 100% *program API* coverage. Furthermore, for networking software, local test harnesses do not suffice, requiring elaborate setups involving multiple software components.

Our work seeks to counter practical limitations of fuzz testing using a complementary approach. It builds on the idea that the reciprocal nature of static analysis and fuzzing may be leveraged to increase the effectiveness of source-code security audits. Our key insight is that vulnerabilities discovered using a fuzzer can be localized to a

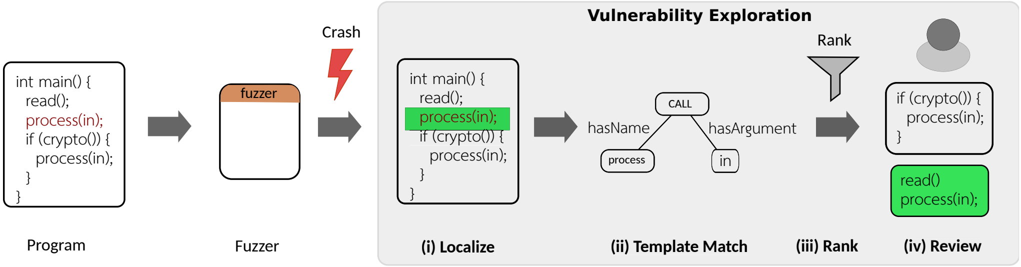


Fig. 5.1.: Work-flow of static vulnerability exploration. Templates generated from fault localized code are used to find recurring instances of a fuzzer-discovered vulnerability. The resulting matches are ranked to focus attention on potential recurring vulnerabilities in untested code.

small portion of application code from which vulnerability templates may be derived. These templates may then be used to find recurring vulnerabilities that may have been either missed by the fuzzer, or are present in code portions that lack a fuzzable test harness.

Leveraging static analysis to complement fuzzing is appealing for two reasons. First, static analysis does not require a test harness, making it well-suited for our problem setting. Second, by taking a program-centric view, static analysis provides a greater overall assurance on software quality or lack thereof. Moreover, since we leverage concrete test cases to bootstrap our analysis, our vulnerability templates focus on a specific fault pattern that has occurred at least once with a demonstrable test input. This begets greater confidence in the returned matches and a higher tolerance for false positives, from an analyst's point of view.

The proposed vulnerability exploration framework requires a coupling between dynamic and static analysis. We begin by fuzzing a readily available program test case. Subsequently, the following steps are taken to enable static exploration of fuzzer-determined program crashes.

- **Fault localization**: We localize vulnerabilities (faults) reported by the fuzzer to a small portion of the code base using either a dynamic memory error detector such as AddressSanitizer [135], or using differential execution slices. Fault localization serves as an interface for coupling dynamic and static analyses, and facilitates automatic generation of vulnerability templates.

- **Vulnerability Templates**: Using lines of code returned by the fault localization module, together with the crash stack trace, we automatically generate vulnerability templates. The templates are encoded using code properties based on a program abstraction such as the abstract syntax tree (AST). Template matching is used to for finding potentially recurring vulnerabilities.

Algorithm 3 Pseudocode for execution slice based fault localization.

```
function OBTAIN-SLICE($Input, Program$)
    ▷ Slice generated using coverage tracer
    Return lines executed by $Program(Input)$

function OBTAIN-DICE($Slice1, Slice2$)
    dice = $Slice1$ - $Slice2$
    return dice

function LOCALIZE-FAILURE($Fault - Input, Program, Fuzz - Corpus$)
    fault-slice = obtain-slice($Fault - Input, Program$)
    nonfault-input = obtain-parent-mutation($Fault - Input, Fuzz - Corpus$)
    nonfault-slice = obtain-slice($nonfault - input, Program$)
    fault-dice = obtain-dice(fault-slice, nonfault-slice)
    return fault-dice
```

- **Ranking Matches**: We rank matches returned by template matching before it is made available for human review. Matches comprising lines of code not covered by fuzzing are ranked higher than those that have already been fuzzed.

- **Validation:** Finally, we manually audit the results returned by our analysis framework to ascertain if they can manifest as vulnerabilities in practice.

5.2.1 Fault Localization

Although a program stack trace indicates where a *crash* happened, it does not necessarily pin-point the root-cause of the failure. This is because, a failure (e.g., memory access violation) manifests much after the trail of the faulty program instructions has been erased from the active program stack. Therefore, fault localization is crucial for templating the root-cause of a vulnerability.

We localize a fuzzer-discovered program failure using a memory detector such as AddressSanitizer [135]. AddressSanitizer is a dynamic analysis tool that keeps track of the state of use of program memory at run time, flagging out-of-bounds reads/writes at the time of occurrence. However, AddressSanitizer cannot localize failures *not* caused by memory access violations. For this reason, we additionally employ a differential execution slicing [1] algorithm to localize general-purpose defects.

---

[1] An execution slice is the set of source lines of code/branches executed by a given input.

Agrawal et al. [1] first proposed the use of differential execution slices (that the authors named execution dices) to localize a general-purpose program fault. Algorithm 3 shows an overview of our implementation of this technique. First, the execution slice for a faulty input is obtained ($fault - slice$, line 10 of Algorithm 3). Second, the fuzzer mutation that preceded the faulty input and did not lead to a failure is determined (line 11), and the execution slice for this input obtained (line 12). Finally, the set difference of the faulty and the non-faulty execution slices is obtained (line 13). This set difference is called the fault dice for the observed failure. We obtain execution slices of a program using the SanitizerCoverage tool [28].

In summary, fault localization helps us localize a fuzzer-discovered vulnerability to a small portion of the codebase. Faulty code may then be used to automatically generate vulnerability templates.

5.2.2 Vulnerability Templates

Faulty code snippets contain syntactic and semantic information pertaining to a program failure. For example, the fact that dereference of the `len` field from a pointer to `struct udp` leads to an out-of-bounds memory access contains (i) the syntactic information that `len` field dereference of a data-type `struct udp` are potentially error-prone; and (ii) the semantic information that tainted input flows into the `struct udp` type record, and that appropriate sanitization is missing in this particular instance. Therefore, we leverage both syntactic, and semantic information to facilitate static exploration of fuzzer-determined program crashes.

Syntactic and semantic templates are derived from localized code snippets, and the crash stack trace. Syntactic templates are matched against the program's abstract syntax tree (AST) representation, while semantic templates against the program's control flow graph (CFG) representation. In the following, we briefly describe how templates are generated, and subsequently matched.

Syntactic Templates Syntactic templates are matched against the program abstract syntax tree (AST). They may be formulated as functional predicates on properties of AST nodes. We describe the process of formulating and matching AST templates using an out-of-bounds read in UDP parsing code of Open vSwitch v2.6.1 that was found by afl-fuzz and AddressSanitizer.

Listing 5.2 shows the code snippet responsible for the out-of-bounds read. The faulty read occurs on line 636 of Listing 5.2 while dereferencing the `udp_header`

```
624 static inline bool
625 check_l4_udp(const struct conn_key *key,
626     const void *data, size_t size,
627     const void *l3)
628 {
629  const struct udp_header *udp = data;
630
631 - // Bounds check on data size missing
632 - if (size < UDP_HEADER_LEN) {
633 -   return false;
634 - }
635
636  size_t udp_len = ntohs(udp->udp_len);
637
638  if (OVS_UNLIKELY(udp_len < UDP_HEADER_LEN ||
639   udp_len > size)) {
640   return false;
641  }
642  ...
643 }
```

Listing 5.2: Code snippet from Open vSwitch v2.6.1 that contains a buffer overread vulnerability in UDP packet parsing code.

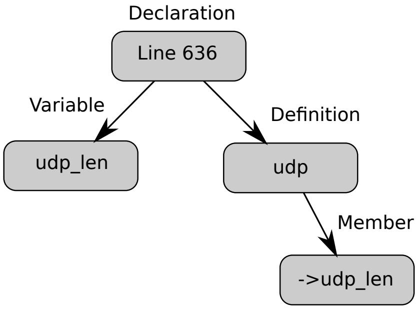


Fig. 5.2.: AST of the localized fault that triggers an out-of-bounds read in UDP packet parsing code.

struct field called `udp_len`. The stack trace provided by AddressSanitizer is shown in Listing 5.3. In this instance, the fault is localized to the function named `check_l4_udp`. Post fault localization, a vulnerability (AST) template is derived from the AST of the localized code itself.

Figure 5.2 shows the AST fragment of the localized faulty code snippet, generated using the Clang compiler. The AST fragment is a sub-tree rooted at the declaration statement on line 636, that assigns a variable named `udp_len` of type `size_t`, to the value obtained by dereferencing a struct field called `udp_len` of type `const unsigned short` from a pointer named `udp` that points to a variable of type to `struct udp_header`. Using the filtered AST fragment, we use AST template matching to find similar declaration statements where `udp_len` is dereferenced. The templates are generated by automatically parsing the AST fragment (as shown in Figure 5.2), and creating Clang libASTMatcher [27] style functional predicates.

```
================================================
==48662==ERROR: AddressSanitizer:
heap-buffer-overflow on address 0x60600000ef3a at
 pc 0x0000005fd716 bp 0x7ffddc709c70
 sp 0x7ffddc709c68

READ of size 2 at 0x60600000ef3a thread T0
    #0 0x5fd715 in check_l4_udp
    lib/conntrack.c:636:33
    #1 0x5fcdcb in extract_l4
    lib/conntrack.c:903:29
    #2 0x5f84bd in conn_key_extract
    lib/conntrack.c:978:13
    #3 0x5f78c4 in conntrack_execute
    lib/conntrack.c:304:14
    #4 0x56df58 in pcap_batch_execute_conntrack
    tests/test-conntrack.c:186:9
    ...
================================================
```

Listing 5.3: Stack trace for the buffer overread in UDP packet parsing code obtained using AddressSanitizer.

Subsequently, template matching is done on the entire codebase. Listing 5.5 shows the generated template and the matches discovered.

AST templates are superior to simple code searching tools such as `grep` for multiple reasons. First, they encode type information necessary to filter through only the relevant data types. Second, they are flexible enough to mine for selective code fragments, such as searching for `udp_len` dereferences in binary operations in addition to declaration statements only.

Listing 5.4 shows one of the matches discovered (see Match #3 of Listing 5.5). In the code snippet shown in Listing 5.4, the OVS controller function named `pinctrl_handle_put_dhcpv6_opts` handles an incoming DHCP packet (containing a UDP packet) that is assigned to a pointer to `struct udp_header`, and subsequently dereferenced in the absence of a bounds-check on the length of the received packet. This is one of the bugs found using syntactic template matching that was reported upstream, and subsequently patched by the vendor [123]. Moreover, this match alerted the OvS developers to a similar flaw in the DNS header parsing code.

To be precise, vulnerability templates need to encode both data and control flow relevant failure inducing code. Otherwise, explicit sanitization of tainted input will be missed, leading to false positives. To this end, we augment syntactic template matching with semantic (control and data-flow) template matching.

Semantic Templates Control and data-flow templates encode semantic code properties needed to examine the flow of tainted input. However, since each defect is

```
538 static void
539 pinctrl_handle_put_dhcpv6_opts(struct dp_packet
540  *pkt_in, struct ofputil_packet_in *pin,
541  struct ofpbuf *userdata, struct ofpbuf
542 *continuation OVS_UNUSED)
543 {
544 ...
545     // Incoming packet parsed into udp struct
546     struct udp_header *in_udp =
547       dp_packet_l4(pkt_in);
548 ...
549     // Dereference missing bounds checking
550     size_t udp_len = ntohs(in_udp->udp_len);
551
552 ...
553 }
```

Listing 5.4: Match returned using automatically generated AST template shows a potentially recurring vulnerability in Open vSwitch 2.6.1. This new flaw was present in the portion of OvS code that lacked a test harness and was found during syntactic template matching.

```
=============================== Query ===============================

let member memberExpr(allOf(hasDeclaration(namedDecl(hasName("udp_len"))),
      hasDescendant(declRefExpr(hasType(pointsTo(
      recordDecl(hasName("udp_header"))))))))

m declStmt(hasDescendant(member))

=============================== Matches ===============================

Match #1:

ovn/controller/pinctrl.c:635:5: note: "root" binds here
    out_ip6->ip6_ctlun.ip6_un1.ip6_un1_plen = out_udp->udp_len;
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Match #2:

tests/test-conntrack.c:52:9: note: "root" binds here
        udp->udp_src = htons(ntohs(udp->udp_src) + tid);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Match #3:

ovn/controller/pinctrl.c:550:5: note: "root" binds here
    size_t udp_len = ntohs(in_udp->udp_len);
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3 Matches.
```

Listing 5.5: AST template matching and its output. The code snippet surrounding match #3 is shown in Listing 5.4.

Algorithm 4 Pseudocode for ranking statically explored vulnerability matches.

```
function isHigh($Matching - unit, Coverset$)

    for each $m - unit$ in $Coverset$ do
        if $m - unit == Matching - unit$ then return True
    return False

function rank-matches($Matches, Coverset$)
    RHigh = $\emptyset$
    RLow = $\emptyset$

    for each $match$ in $Matches$ do
        if isHigh($match, Coverset$) then
            RHigh $+=$ $match$
        else
            RLow $+=$ $match$
    return (RHigh, RLow)
```

characterized by unique control and data-flow, semantic templates are harder to automate. We remedy this problem by providing *fixed* semantic templates that are generic enough to be applied to any defect type.

We parse the program crash stack trace to perform semantic template matching. First, we determine the function in which the program fails (top-most frame in the crash trace), and generate a template to match other call-sites of this function. We call this a *callsite* template. Callsite templates intuitively capture the insight that, if a program failure manifests in a given function, other calls to that function demand inspection. Second, for memory access violation related vulnerabilities, we determine the data-type of the variable that led to an access violation, and assume that this data-type is *tainted*. Subsequently, we perform taint analysis on this data-type terminating at pre-determined security-sensitive sinks such as `memcpy`, `strcpy` etc. We call this a *taint* template. Taint templates provide insight on risky usages of a data-type that is known to have caused a memory access violation. Callsite and taint templates are matched against the program control flow graph (CFG). They have been implemented as extensions to the Clang Static Analyzer framework [90].

5.2.3 Match Ranking

Matches returned using static template matching may be used to (in)validate potentially recurring vulnerabilities in a codebase. However, since vulnerability templates over-approximate failure-inducing code patterns, false positives are inevitable. We remedy the false-positive problem using a simple yet practical match ranking algorithm.

Algorithm 4 presents the pseudocode for our match ranking algorithm. The procedure called `RANK-MATCHES` accepts the set of template matches (denoted as $Matches$), and the set of program functions covered by fuzz testing (denoted as $Coverset$) as input, and returns a partially orders list suitable for manual review. For each match, we apply a ranking predicate on the program function in which the match was found. We call this function, the *matching unit*. The ranking predicate (denoted as the procedure $isHigh$) takes two input parameters: the matching function name, and the $Coverset$. Under the hood, $isHigh$ simply performs a test of set membership; it checks if the matching unit is a member of the coverset, returning `True` if it is a member, `False` otherwise. All matching units that satisfy the ranking predicate are ranked high, while the rest are ranked low. The ranked list is returned as output.

Our ranking algorithm is implemented in Python using a hash table based data structure. Therefore, ranking a match takes $O(1)$ on average, and $O(n)$ in the worst case, where n is the number of functions in the coverset. On average, the time to rank all matches grows linearly with the number of matches. This is really fast in practice e.g., in the order of a few milliseconds (see Table 5.3).

Our prototype leverages *GCov* [53], a publicly available program coverage tracing tool, for obtaining the coverset of a fuzzer corpus. Although our prototype currently uses function as a matching unit, it may be suitably altered to work at the level of source line of code (basic blocks). However, given that a fuzzer corpus typically contains thousands of test cases, we chose function-level tracing for performance reasons.

5.2.4 Validation

Although match ranking helps reduce the burden of false positives, it does not eliminate them entirely. Therefore, we rely on manual audit to ascertain the validity of analysis reports. Nonetheless, our approach focuses attention on recurrences of demonstrably vulnerable code patterns, thereby reducing the extent of manual code audit.

5.3 Evaluation

Fuzzer-Discovered Vulnerability	CVE ID	Explored Matches	True Positives
Out-of-bounds read (IP)	CVE-2016-10377 [37]	5	0
Out-of-bounds read (TCP)	CVE-2017-9264 [40]	10	0
Out-of-bounds read (UDP)	CVE-2017-9264	2	1
Out-of-bounds read (IPv6)	CVE-2017-9264	3	0
Remote DoS due to assertion failure	CVE-2017-9214 [38]	22	0
Remote DoS due to unhandled packet	CVE-2017-9263 [39]	34	0
Out-of-bounds read	CVE-2017-9265 [41]	1	0
Total		96	1

Tab. 5.1.: Summary of static vulnerability exploration carried out on vulnerabilities found by fuzzing Open vSwitch. For each fuzzer-discovered vulnerability, our prototype generated a vulnerability template, and matched it against the entire codebase.

CVE ID	Explored matches	Ranked high (untested)	Reduction in FP (in %)
CVE-2016-10377	5	0	100
CVE-2017-9264	10	0	100
CVE-2017-9264	2	2	0
CVE-2017-9264	3	0	100
CVE-2017-9214	41	17	59
CVE-2017-9263	34	17	50
CVE-2017-9265	1	0	100
Total	96	36	62

Tab. 5.2.: Effectiveness of our matching ranking algorithm in highlighting untested code, and assisting in fast review of matches.

We evaluated our approach on multiple versions of Open vSwitch, an open-source virtual switch used in data centers. We chose Open vSwitch for evaluation because (i) it is a good representative of production code; (ii) it has insufficient test harnesses suitable for fuzzing, resulting in program edge coverage of less than 5%.

Our evaluations were performed using afl-fuzz for fuzzing, AddressSanitizer for fault localization, falling back to our implementation of differential slice-based fault localization, and our implementation of static template generation, matching, and ranking algorithms. Experiments were carried out on a 64-bit machine with 80 CPU threads (Intel Xeon E7-4870) clocked at 2.4 GHz, and 512 GB RAM.

Fuzzing and Fault Localization Using the baseline fuzzer, we discovered multiple out-of-bounds reads and assertion failures in packet parsing code in Open vSwitch. All the discovered flaws were triaged to ascertain their security impact, and subsequently reported upstream and fixed. For each unique vulnerability, we used our fault localization module comprising AddressSanitizer, and differential execution slicing, to determine the lines of code triggering the vulnerability.

Template Matching Using localized code, we automatically generated a template suitable for matching similar code patterns elsewhere in the codebase. For example, the AST snippet shown in Figure 5.2 was parsed to derive a template for CVE-2017-9264. Subsequently, we used the tool `clang-query` to perform template matching using the derived template. Listing 5.5 shows the outcome of template matching for one of the bugs comprising CVE-2017-9264. For each vulnerability that the fuzzer discovered, we counted the number of matches (excluding the known vulnerability itself) returned using template matching.

We used semantic template matching only when syntactic template matching was too broad to capture the code pattern underlying the vulnerability. For example, if a program crash was caused by a failed assertion, syntactic templates (that matched calls to all assertion statements), were augmented with semantic templates (that matched a smaller subset of assertion statements involving tainted data types).

Ranking The returned matches were ranked using our proposed ranking algorithm (see Algorithm 4), and the ranked output was used as a starting point for manual security audit. Matches ranked high were reviewed first. This enabled us to devote more time to audit untested code, than the code that had already undergone testing.

5.3.1 Analysis Effectiveness

We evaluated the effectiveness of our approach in two ways: Quantifying (i) the raw false positive rate of our analysis; (ii) the benefit of the proposed ranking algorithm in reducing the effective false positive rate after match ranking was done.

To quantify the number of raw false positives, we counted the total number of statically explored matches, and the number of true positives among them. A match was deemed a true positive if manual audit revealed that the tainted instruction underwent no prior sanitization and was thus potentially vulnerable. Table 5.1

CVE ID	Localization	Syntactic	Semantic	Ranking	Total Run Time	Normalized
CVE-2016-10377	82ms	1.66s	–	63ms	1.80s	0.20x
CVE-2017-9264 (TCP)	84ms	3.20s	–	64ms	3.34s	0.25x
CVE-2017-9264 (UDP)	86ms	4.77s	–	59ms	4.91s	0.37x
CVE-2017-9264 (IPv6)	91ms	4.71s	–	60ms	4.86s	0.36x
CVE-2017-9214	9ms	8.44s	44.17s	60ms	52.67s	5.51x
CVE-2017-9263	9ms	11.88s	44.26s	59ms	57.09s	5.97x
CVE-2017-9265	111ms	5.74s	–	56ms	5.9s	0.62x

Tab. 5.3.: Run times of fault localization, template matching, and match ranking for all statically explored vulnerabilities in Open vSwitch. The absolute and relative (to code compilation) run times for our end-to-end analysis is presented in the final two columns. A normalized run time of 2x denotes that our end-to-end analysis takes twice as long as code compilation.

summarizes our findings. Our prototype returned a total of 96 matches for the 7 vulnerabilities found by fuzzing (listed in column 1 of Table 5.1). Out of 96 matches, only one match corresponding to CVE-2017-9264 was deemed a new potential vulnerability. This was reported upstream and subsequently patched [123]. Moreover, the reported (potential) vulnerability helped OvS developers uncover another similar flaw in the DHCPv6 parsing code that followed the patched UDP flaw.

Our ranking algorithm ranked untested code over tested code, thereby helping reduce the manual effort involved in validating potential false positives. Although it is hard to correctly quantify the benefit of our ranking algorithm in bringing down the false positive rate, we employ a notion of *effective* false positive rate. We define the effective false positive rate to be the false positive rate only among highly ranked matches. This is intuitive, since auditing untested code is usually more interesting to a security analyst than auditing code that has already undergone testing. Table 5.2 summarizes the number of effective false positives due to our analysis. In total, there were 36 matches (out of 96) that were ranked high, bringing down the raw false positive rate by 62%. Naturally, we confirmed that the single true positive was among the highly ranked matches.

Match ranking helps reduce, but not eliminate the number of false positives. Indeed, 1 correct match out of 36 matches is very low. Having said that, our approach has borne good results in practice, and has helped advance the tooling required for secure coding. The additional patch that our approach contributed to is not the only way in which our approach met this objective. We discovered that the template derived from the vulnerability CVE-2016-10377 present in an earlier version of Open vSwitch (v2.5.0), could have helped eliminate a similar vulnerability (CVE-2017-9264) in a later version (v2.6.1), perhaps during software development itself. This shows that our approach is suitable for regression testing. Indeed, OvS developers noted in personal communications with the authors that the matches returned by our tooling not only encouraged reasoning about corner cases in software development, but helped catch bugs (latent vulnerabilities) at an early stage.

5.3.2 Analysis Runtime

We quantified the run time of our tooling by measuring the total and constituent run times of our work-flow steps, starting from fault localization, and template matching, to match ranking. Table 5.3 presents our analysis run times for each of the fuzzer-discovered vulnerabilities in Open vSwitch. Since fault localization was done using dynamic tooling (AddressSanitizer/coverage tracing), it was orders of magnitude faster (ranging between 9–111 milliseconds) than the time required for static template matching. For each fuzzer-discovered vulnerability, we measured the template matching run time as the time required to construct and match the vulnerability template against the entire codebase. Template matching run time comprised between 92–99% of the end-to-end runtime of our tooling, and ranged from 1.8 seconds to 57.09 seconds. Syntactic template matching was up to 4x faster than semantic template matching. This conformed to our expectations, as semantic matching is slower due to the need to encode (and check) program data and control flow in addition to its syntactic properties. Nonetheless, our end-to-end vulnerability analysis had a normalized run time (relative to code compilation time) of between 0.2x to 5.97x. The potential vulnerability that our analysis pointed out in untested UDP parsing code, was returned in roughly a third of the time taken for code compilation of the codebase. This shows that our syntactic analysis is fast enough to be applied on each build of a codebase, while our semantic analysis is more suitable to be invoked during daily builds. Moreover, given the low run time of our analysis, templates derived from a vulnerability discovered in a given release may be continuously applied to future versions of the same codebase as part of regression testing.

5.4 Related Work

Our work brings together ideas from recurring vulnerability detection, and program analysis and testing. In the following paragraphs, we compare our work to advances in these areas.

Patch-based Discovery of Recurring Vulnerabilities Redebug [73] and Securesync [118] find recurring vulnerabilities by using syntax matching of templates derived from vulnerability patches. Thus, *patched* vulnerabilities form the basis of their template-based matching algorithms. In contrast, we template a vulnerability based on automatically localized failures, and debug information obtained from fuzzer reported crashes. What makes our setting more challenging is the lack of a reliable code pattern (usually obtained from a patch) to build a template from. As we have shown, it is possible to construct vulnerability templates even in this constrained environment and find additional vulnerabilities even *in the absence* of patches.

Code Clone Detection We are not the first to present a pattern-based approach to vulnerability detection. Yamaguchi et al. [161] project vulnerable code patterns derived from patched vulnerabilities on to a vector space. This permits them to extrapolate known vulnerabilities in current code, thereby permitting the discovery of recurring vulnerabilities.

Other researchers have focused on finding code clones regardless of them manifesting as vulnerabilities [13, 12, 82, 94]. Code clone detection tools such as CPMiner [87], CCFinder [78], Deckard [74] solve the problem of finding code clones but rely on sample code input to be provided. These tools solve the more general problem of finding identical copies of user-provided code. Although these tools serve as a building block for recurring vulnerability discovery, they require that the user specifies the code segment to be matched. In a setting where a security analyst is auditing third-party code, manual specification of code templates might not be feasible. By automatically performing template matching on fuzzer-discovered program crashes, leverage the fuzzer for the specification for vulnerable code patterns.

Hybrid Vulnerability Discovery SAGE [60] is a white-box fuzz testing tool that combines fuzz testing with dynamic test-case generation. Constraints accumulated during fuzz testing are solved using an SMT solver to generate test cases that the fuzzer alone could not generate. This is expensive because it requires a sophisticated solver. In a similar vein, Driller [145] augments fuzzing through selectively resorting to symbolic execution when fuzzer encounters coverage bottlenecks. The use of symbolic execution to augment fuzzing is complementary to our approach. In practice, security audits would benefit from both our approach as well as that proposed by prior researchers.

Saner [11] combines static and dynamic analyses towards identifying XSS and SQL injection vulnerabilities in web applications. The authors of Saner use static analysis to capture a set of taint source-sink pairs from web application code, and subsequently use dynamic analysis on the captured pairs to tease out vulnerabilities. Their evaluation on popular PHP applications show that dynamic analysis is able to bring down the number of false positives produced by static analysis, and find multiple vulnerabilities. Like our work, Saner demonstrates that static and dynamic analyses can effectively complement each other. In contrast to Saner, we differ in the order of analyses performed (we perform static analysis driven vulnerability exploration after confirmed taint source-sink pairs have been found), and in the target programming language.

Yamaguchi et al. [159] automatically infer search patterns for taint-style vulnerabilities from source code by combining static analysis and unsupervised machine learning. They show that their approach helps reduce the amount of code audit necessary to spot recurring vulnerabilities by up to 94.9%, enabling them to find 8 zero-day vulnerabilities in production software. Their work is close in spirit to ours. However, we avoid the computational overhead involved in their workflow (building a code property graph, pattern clustering etc.), while retaining their template matching run time. In our framework, fault localization and result ranking run times are almost negligible.

6

Conclusion

> *Ultimately, the key to winning the hearts and minds of practitioners is very simple: you need to show them how the proposed approach finds new, interesting bugs in the software they care about.*
>
> – Michał Zalewski, *Symbolic execution in vuln research*

Vulnerabilities in shipped code increase the risk of active harm. At the same time, eliminating vulnerabilities is a hard problem because it is almost impossible to anticipate all security-relevant corner cases in complex software. Therefore, an early diagnosis of software vulnerabilities is crucial for increasing our confidence in software.

This thesis has introduced *compiler assisted vulnerability assessment*, a set of methods based on compiler-based static analysis that advances the state-of-the-art in vulnerability assessment of production software. Our methods are practical, and can be incorporated either at the development or the testing stage of the software development lifecycle, permitting early vulnerability diagnosis. Moreover, the developed techniques address both object-oriented software and network parsers, showing that our techniques not only scale up to large sized codebases but also work for different application classes.

The methods presented in this thesis showcase the adaptable nature of static analysis for vulnerability diagnosis. Our methods leverage program analysis both traditionally (accept source code as input and produce bug reports, see Chapter 3), and less traditionally to infer input format (Chapter 4), and for vulnerability template matching (Chapter 5). Our vulnerability assessment tools have been used to diagnose tens of zero-day vulnerabilities in production software.

In the following, we briefly highlight the main results of this dissertation. Finally, we outline directions for future work.

6.1 Summary of Results

This dissertation began with a manual security audit of an emerging web based OS called Firefox OS. Subsequently, we proposed three compiler-assisted methods to scale up vulnerability discovery. To this end, we designed and implemented vulnerability assessment tools that can be deployed either during software development or during the quality assurance stage. Although the first of our methods studies static vulnerability diagnosis, the latter two methods exploit the complementary nature of dynamic and static approaches in novel ways. The last of our methods closes the loop by showing that static vulnerability diagnosis can be (re)invoked to increase our confidence on the findings from the program testing stage. In specific terms, this dissertation makes the following contributions.

Security Audit of Firefox OS We are witnessing a convergence of the web and smartphone technologies in emerging systems such as Firefox OS and Chrome OS. We lack an understanding of new classes of vulnerabilities that may be introduced due to this convergence. To this end, we have presented the first security audit of Firefox OS to the best of our knowledge. Using manual source code audits and a penetration testing approach, we show that (i) lack of code integrity is a cause for concern in the web provisioning model adopted by Mozilla, and (ii) insufficient identification of security principals may lead to subtle user experience related security issues.

A method for diagnosing distributed vulnerabilities. Due to the scale of today's software development, vulnerabilities may only manifest when two or more subsystems interact in a specific manner. To this end, we have presented a method for diagnosing distributed vulnerabilities that is scalable to large codebases. The method builds on a novel event collection system, and demand-driven whole-program analyzer to perform vulnerability diagnosis in a staged manner. Using this design, we have been able to scalably diagnose vulnerabilities spanning source files in codebases such as Chromium and Firefox that contain millions of lines of code. Based on empirical evaluation of our method, we have demonstrated that our analyses is reasonably fast, cheap to deploy, and capable of flagging vulnerabilities in production code (chapter 3).

A method for inferring input format specification. Exhaustive testing of parsing applications is challenging owing to a large number of program paths involved. We have presented a static analysis based approach to identify input format specification of a parser by analyzing source code. We use syntactic and semantic code analysis to extract regular and non-regular portions of the input specification. Using a case study approach, we have shown that our approach improves test coverage by 10–15%, and uncovers 15 zero-day vulnerabilities in network parsers that are not found by standalone fuzzing (chapter 4).

A method for exploring fuzzer discovered vulnerabilities. Legacy code is insufficiently stress tested because of a lack of security test cases. We have presented an approach to guide security code audits in this scenario. Our approach builds on the idea that functional test cases can serve as a starting point to initiate an exploration of the vulnerability landscape of an application. To this end, we build a static analysis tool that, given a confirmed fault pattern, returns similar matches across the code base. We have demonstrated our method's utility by using it for security audit of a popular virtual switch called Open vSwitch. We show that our method is useful not only for vulnerability discovery but also regression testing (chapter 5).

6.2 Future Directions

We now outline directions for future work by listing some ideas for progress.

Data Protection Violation Analysis In this dissertation, we focus on methods for diagnosing memory corruption vulnerabilities for systems code. Although this is an important vulnerability class, privacy violations are going to assume importance in the near future. One emerging class of vulnerabilities is violation of security policies mandated by legislation. For example, the upcoming European Union's general data protection regulation (GDPR) [150] requires software vendors to enforce strict data usage guidelines. Automated analysis is going to be one of the enablers of such policy enforcement and validation. Extending the methods in this dissertation and applying them to data violations is one avenue for future research.

Security Analysis of Modern Software Although C/C++ is still popular in the embedded systems domain and for writing performant cross-platform code, programming languages such as JavaScript and Python address the needs of modern software. The GitHub 2017 year in review [71] shows that these were the three most actively used programming languages, going by the number of pull requests issued. Applying compiler-based vulnerability assessment techniques to modern software poses interesting challenges. One specific problem of interest is to make analysis language independent so that security violations can be identified across-the-board.

Machine Learning Assisted Program Analysis With increased availability of dedicated hardware, machine learning is starting to influence the domains of program analysis and security testing [125, 58]. As we have seen in Chapter 5, vulnerability patterns can drive the search for recurring vulnerabilities. Machine learning is well-suited to assist pattern-based vulnerability analysis in two ways. It can learn characteristics of program input and even generate new inputs based on the learning phase. This is of particular use in fuzz testing because learning algorithms provide a structured way to provide good starting inputs (*seeds*) to fuzz testing tools. Moreover, as the state-of-the-art in machine learning based vulnerability detection shows [159, 160], machine learning may also be used to infer security policy from source code to drive program analysis. Advancing these directions has the potential to significantly impact the practice of vulnerability assessment.

A

Reported Vulnerabilities

Table A.1 lists vulnerabilities discovered during the course of this thesis. All vulnerabilities were responsibly disclosed to the vendor, following which a common vulnerabilities and exposures (CVE) identifier was assigned to them by MITRE.

Software	Vulnerability Description	CVE ID
Open vSwitch	Out-of-bounds (OOB) write in MPLS parser permits remote code execution	CVE-2016-2074
	OOB read in TCP/UDP/IPv6 parsers	CVE-2017-9264
	OOB read in flow parser	CVE-2016-10377
	OOB read in OpenFlow (OF) parser	CVE-2017-9214
	Denial of service (DoS) due to unhandled case in OF parser	CVE-2017-9263
	OOB read in OF parser	CVE-2017-9265
GNU oSIP2	DoS due to OOB write in SIP parser	CVE-2017-7853
	OOB write in SIP parser	CVE-2016-10324
	DoS due to OOB write in SIP parser	CVE-2016-10325
	DoS due to OOB write in SIP parser	CVE-2016-10326
Snort++	DoS due to logical error in Ethernet parser	CVE-2017-6657
	OOB read in packet decoder	CVE-2017-6658
tcpdump	OOB read in utility function	CVE-2017-13011
	OOB read in the ICMP parser	CVE-2017-13012
	OOB read in the ARP parser	CVE-2017-13013
	OOB read in the EAP parser	CVE-2017-13015
	OOB read in the ISO ES-IS parser	CVE-2017-13016
	OOB read in the DHCPv6 parser	CVE-2017-13017
	OOB read in the PGM parser	CVE-2017-13018
	OOB read in the PGM parser	CVE-2017-13019
	OOB read in the VTP parser	CVE-2017-13020
	OOB read in the ICMPv6 parser	CVE-2017-13021
	OOB read in the IP parser	CVE-2017-13022
	OOB read in the IPv6 mobility parser	CVE-2017-13023
	OOB read in the IPv6 mobility parser	CVE-2017-13024
	OOB read in the IPv6 mobility parser	CVE-2017-13025
	OOB read in the ISO IS-IS parser	CVE-2017-13026
	OOB read in the LLDP parser	CVE-2017-13027
	OOB read in the BOOTP parser	CVE-2017-13028
	OOB read in the PPP parser	CVE-2017-13029
	OOB read in the PIM parser	CVE-2017-13030
	OOB read in the IPv6 fragmentation header parser	CVE-2017-13031
	OOB read in the RADIUS parser	CVE-2017-13032
	OOB read in the VTP parser	CVE-2017-13033
	OOB read in the PGM parser	CVE-2017-13034
	OOB read in the ISO IS-IS parser	CVE-2017-13035
	OOB read in the OSPFv3 parser	CVE-2017-13036
	OOB read in the IP parser	CVE-2017-13037
	OOB read in the ISAKMP parser	CVE-2017-13039
	OOB read in the HNCP parser	CVE-2017-13042
	OOB read in the BGP parser	CVE-2017-13043
	OOB read in the HNCP parser	CVE-2017-13044
	OOB read in the OLSR parser	CVE-2017-13688

Tab. A.1.: Vulnerabilities found during the course of this dissertation. This table continues on the next page.

Software	Vulnerability Description	CVE ID
tcpdump	OOB read in the VQP parser	CVE-2017-13045
	OOB read in the BGP parser	CVE-2017-13046
	OOB read in the ISO ES-IS parser	CVE-2017-13047
	OOB read in the RSVP parser	CVE-2017-13048
	OOB read in the RPKI-Router parser	CVE-2017-13050
	OOB read in the IKEv1 parser	CVE-2017-13689
	OOB read in the IKEv2 parser	CVE-2017-13690
	OOB read in the RSVP parser	CVE-2017-13051
	OOB read in the ISO IS-IS parser	CVE-2017-13055
	OOB read in the CFM parser	CVE-2017-13052
	OOB read in the BGP parser	CVE-2017-13053
	OOB read in the LLDP parser	CVE-2017-13054

List of Figures

List of Tables

Listings

Bibliography

[1] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. “Fault localization using execution slices and dataflow tests”. In: *Proc. IEEE International Symposium on Software Reliability Engineering*. 1995, pp. 143–151 (cit. on p. 72).

[2] Devdatta Akhawe and Adrienne Porter Felt. “Alice in Warningland: A Large-scale Field Study of Browser Security Warning Effectiveness”. In: *Proceedings of the 22Nd USENIX Conference on Security*. SEC’13. Washington, D.C.: USENIX Association, 2013, pp. 257–272 (cit. on p. 24).

[3] Frances E. Allen. “Control Flow Analysis”. In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19 (cit. on p. 5).

[4] Chaitrali Amrutkar, Patrick Traynor, and Paul C. van Oorschot. “An Empirical Evaluation of Security Indicators in Mobile Web Browsers”. In: *IEEE Transactions on Mobile Computing* PrePrints (2013) (cit. on p. 24).

[5] James P Anderson. “Computer security threat monitoring and surveillance”. In: *Technical Report, James P. Anderson Company* (1980) (cit. on p. 2).

[6] K. Ashcraft and D. Engler. “Using programmer-written compiler extensions to catch security holes”. In: *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*. 2002, pp. 143–159 (cit. on p. 31).

[7] A. Austin and L. Williams. “One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques”. In: *2011 International Symposium on Empirical Software Engineering and Measurement*. Sept. 2011, pp. 97–106 (cit. on p. 27).

[8] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. “AEG: Automatic Exploit Generation.” In: *NDSS*. Vol. 11. 2011, pp. 59–66 (cit. on p. 43).

[9] David F. Bacon and Peter F. Sweeney. “Fast Static Analysis of C++ Virtual Function Calls”. In: *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’96. San Jose, California, USA: ACM, 1996, pp. 324–341 (cit. on p. 40).

[10] Thomas Ball and Sriram K Rajamani. “The SLAM project: debugging system software via static analysis”. In: *ACM SIGPLAN Notices*. Vol. 37. ACM. 2002, pp. 1–3 (cit. on p. 43).

[11] Davide Balzarotti, Marco Cova, Vika Felmetsger, et al. “Saner: Composing static and dynamic analysis to validate sanitization in web applications”. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 387–401 (cit. on pp. 2, 83).

[12] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. "Clone detection using abstract syntax trees". In: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE. 1998, pp. 368–377 (cit. on p. 82).

[13] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. "Comparison and evaluation of clone detection tools". In: *IEEE Transactions on software engineering* 33.9 (2007) (cit. on p. 82).

[14] Alan W Biermann. "On the inference of Turing machines from sample computations". In: *Artificial Intelligence* 3 (1972), pp. 181–198 (cit. on p. 5).

[15] Matthew A. Bishop. *The Art and Science of Computer Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on p. 3).

[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain". In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. ACM. 2016, pp. 1032–1043 (cit. on pp. 59, 62).

[17] Theodore Book, Adam Pridgen, and Dan S. Wallach. "Longitudinal Analysis of Android Ad Library Permissions". In: *Mobile Security Technologies (MoST 2013)*. 2013 (cit. on p. 18).

[18] Bugzilla@Mozilla. `https://bugzilla.mozilla.org/show_bug.cgi?id=858730` (cit. on p. 19).

[19] *Bugzilla@Mozilla, Bug 1168091*. `https://bugzilla.mozilla.org/show_bug.cgi?id=1168091` (cit. on p. 38).

[20] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis". In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. 2007, pp. 317–329 (cit. on p. 63).

[21] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224 (cit. on pp. 2, 43).

[22] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 265–278 (cit. on p. 2).

[23] *Chromium Issue Tracker, Issue 411177*. `https://code.google.com/p/chromium/issues/detail?id=411177` (cit. on p. 28).

[24] *Chromium Issue Tracker, Issue 411177*. `https://code.google.com/p/chromium/issues/detail?id=411177` (cit. on p. 38).

[25] *Chromium Issue Tracker, Issue 436035*. `https://code.google.com/p/chromium/issues/detail?id=436035` (cit. on p. 38).

[26] Cristina Cifuentes and Bernhard Scholz. “Parfait: designing a scalable bug checker”. In: *Proceedings of the 2008 workshop on Static analysis*. ACM. 2008, pp. 4–11 (cit. on pp. 43, 44).

[27] *Clang AST matcher reference*. `http://clang.llvm.org/docs/LibASTMatchersReference.html`. Accessed: 31/5/2017 (cit. on p. 73).

[28] *Clang/LLVM SanitizerCoverage*. `https://clang.llvm.org/docs/SanitizerCoverage.html`. Accessed: 23/5/2017 (cit. on p. 72).

[29] Clusterfuzzer. *Heap-buffer-overflow in read*. `https://bugs.chromium.org/p/chromium/issues/detail?id=609042`. Accessed: 03/23/17 (cit. on p. 58).

[30] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. “Prospex: Protocol specification extraction”. In: *Proc. IEEE Security & Privacy*. 2009, pp. 110–125 (cit. on pp. 63, 64).

[31] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. *Iterative data-flow analysis, revisited*. Tech. rep. 2004 (cit. on p. 7).

[32] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. *mitmproxy: A free and open source interactive HTTPS proxy*. [Version 3.0]. 2010– (cit. on p. 18).

[33] *Coverity Inc.* `http://www.coverity.com/` (cit. on pp. 5, 43).

[34] Christoph Csallner and Yannis Smaragdakis. “Check’n’crash: combining static checking and testing”. In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 422–431 (cit. on p. 2).

[35] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. “Discoverer: Automatic Protocol Reverse Engineering from Network Traces.” In: *Proc. Usenix Security Symp.* Vol. 158. 2007 (cit. on p. 63).

[36] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. “Tupni: Automatic reverse engineering of input formats”. In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. 2008, pp. 391–402 (cit. on p. 63).

[37] *CVE-2016-10377*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10377`. Accessed: 24/5/2017 (cit. on p. 78).

[38] *CVE-2017-9214*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9214`. Accessed: 24/5/2017 (cit. on p. 78).

[39] *CVE-2017-9263*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9263`. Accessed: 24/5/2017 (cit. on p. 78).

[40] *CVE-2017-9264*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9264`. Accessed: 24/5/2017 (cit. on p. 78).

[41] *CVE-2017-9265*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9265`. Accessed: 24/5/2017 (cit. on p. 78).

[42] *Cyber-attack: Europol says it was unprecedented in scale*. `http://www.bbc.com/news/world-europe-39907965` (cit. on p. 1).

[43] Jeffrey Dean, David Grove, and Craig Chambers. “Optimization of object-oriented programs using static class hierarchy analysis”. In: *ECOOP’95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*. Springer. 1995, pp. 77–101 (cit. on p. 34).

[44] Daniel DeFreez, Bhargava Shastry, Hao Chen, and Jean-Pierre Seifert. “A first look at Firefox OS security”. In: *Mobile Security Technologies*. IEEE. 2014 (cit. on p. ix).

[45] David Evans and David Larochelle. “Improving security using extensible lightweight static analysis”. In: *IEEE software* 19.1 (2002), pp. 42–51 (cit. on p. 2).

[46] Sascha Fahl, Marian Harbach, Thomas Muders, et al. “Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 50–61 (cit. on pp. 18, 24).

[47] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, et al. “Android Permissions: User Attention, Comprehension, and Behavior”. In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. SOUPS ’12. Washington, D.C.: ACM, 2012, 3:1–3:14 (cit. on p. 18).

[48] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. “Permission Re-delegation: Attacks and Defenses”. In: *Proceedings of the 20th USENIX Conference on Security*. SEC’11. San Francisco, CA: USENIX Association, 2011, pp. 22–22 (cit. on p. 18).

[49] Jonathan Foote. *The exploitable GDB plugin*. `https://github.com/jfoote/exploitable`. Accessed: 03/23/17. 2015 (cit. on p. 60).

[50] Jeffrey S Foster, R Johnson, J Kodumal, et al. *CQUAL: A tool for adding type qualifiers to C*. Accessed: 2015-03-26. 2003 (cit. on pp. 43, 44).

[51] *frama-c*. `http://frama-c.com/` (cit. on p. 5).

[52] Keith Brian Gallagher and James R. Lyle. “Using program slicing in software maintenance”. In: *IEEE Transactions on Software Engineering* 17.8 (1991), pp. 751–761 (cit. on p. 51).

[53] *gcov: A test coverage program (Online documentation)*. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. Accessed: 25/5/2017 (cit. on p. 77).

[54] *GeeksPhone*. `http://www.geeksphone.com/` (cit. on p. 18).

[55] Martin Georgiev, Subodh Iyengar, Suman Jana, et al. “The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 38–49 (cit. on pp. 18, 24).

[56] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *ACM SIGPLAN Notices*. Vol. 40. 6. 2005, pp. 213–223 (cit. on p. 2).

[57] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. “Grammar-based whitebox fuzzing”. In: *ACM SIGPLAN Notices*. Vol. 43. 6. 2008, pp. 206–215 (cit. on pp. 2, 64).

[58] Patrice Godefroid, Hila Peleg, and Rishabh Singh. “Learn&Fuzz: Machine Learning for Input Fuzzing”. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 50–59 (cit. on p. 88).

[59] Patrice Godefroid, Michael Y Levin, and David Molnar. “SAGE: whitebox fuzzing for security testing”. In: *ACM Queue* 10.1 (2012), p. 20 (cit. on p. 2).

[60] Patrice Godefroid, Michael Y Levin, and David Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Queue* 10.1 (2012), p. 20 (cit. on p. 83).

[61] Google Inc. *Fuzzer test suite*. `https://github.com/google/fuzzer-test-suite`. Accessed: 03/23/17 (cit. on p. 57).

[62] GrammaTech. *CodeSonar*. `http://www.grammatech.com/codesonar` (cit. on pp. 5, 43).

[63] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. “A System and Language for Building System-specific, Static Analyses”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI ’02. Berlin, Germany: ACM, 2002, pp. 69–82 (cit. on p. 45).

[64] *HAVOC*. `http://research.microsoft.com/en-us/projects/havoc/` (cit. on p. 43).

[65] S. Heelan. “Vulnerability Detection Systems: Think Cyborg, Not Robot”. In: *Security Privacy, IEEE* 9.3 (May 2011), pp. 74–77 (cit. on p. 43).

[66] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. “Software verification with BLAST”. In: *Model Checking Software*. Springer, 2003, pp. 235–239 (cit. on p. 43).

[67] Hewlett Packard. *Fortify Static Code Analyzer*. `http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/` (cit. on pp. 5, 43).

[68] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with Code Fragments.” In: *Proc. Usenix Security Symp.* 2012, pp. 445–458 (cit. on p. 64).

[69] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. 2006 (cit. on p. 51).

[70] CBS Inc. *Global cyberattack strikes dozens of countries, cripples U.K. hospitals*. `https://www.cbsnews.com/news/hospitals-across-britain-hit-by-ransomware-cyberattack/` (cit. on p. 1).

[71] GitHub Inc. *The State of the Octoverse 2017*. `https://octoverse.github.com/` (cit. on p. 87).

[72] Google Inc. *Introducing the Google Chrome OS*. `https://googleblog.blogspot.de/2009/07/introducing-google-chrome-os.html` (cit. on p. 15).

[73] Jiyong Jang, Abeer Agrawal, and David Brumley. “ReDeBug: finding unpatched code clones in entire os distributions”. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 48–62 (cit. on p. 82).

[74] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. “Deckard: Scalable and accurate tree-based detection of code clones”. In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 96–105 (cit. on p. 82).

[75] Xing Jin, Lusha Wang, Tongbo Luo, and Wenliang Du. “Fine-Grained Access Control for HTML5-Based Mobile Applications in Android”. In: *ISC 2013*. 2013 (cit. on p. 24).

[76] S Johnson. “Lint, a C program checker, 1978”. In: *Unix Programmer’s Manual, AT&T Bell Laboratories* () (cit. on p. 42).

[77] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. “Pixy: A static analysis tool for detecting web application vulnerabilities”. In: *Security and Privacy, 2006 IEEE Symposium on*. IEEE. 2006, 6–pp (cit. on p. 2).

[78] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. “CCFinder: a multilinguistic token-based code clone detection system for large scale source code”. In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670 (cit. on p. 82).

[79] Michael F Kleyn and Paul C Gingrich. “GraphTrace—understanding object-oriented systems using concurrently animated views”. In: *ACM Sigplan Notices*. Vol. 23. 11. ACM. 1988, pp. 191–205 (cit. on p. 5).

[80] *Klocwork*. `http://www.klocwork.com/` (cit. on pp. 5, 43).

[81] Jens Knoop and Bernhard Steffen. *Efficient and optimal bit vector data flow analyses: a uniform interprocedural framework*. Inst. für Informatik und Praktische Mathematik, 1993 (cit. on p. 32).

[82] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. “Pattern matching for clone and concept detection”. In: *Reverse engineering*. Springer, 1996, pp. 77–108 (cit. on p. 82).

[83] Ted Kremenek and Dawson Engler. “Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations”. In: *Proceedings of the 10th International Conference on Static Analysis*. SAS’03. San Diego, CA, USA: Springer-Verlag, 2003, pp. 295–315 (cit. on p. 42).

[84] Monica S. Lam, John Whaley, V. Benjamin Livshits, et al. “Context-sensitive Program Analysis As Database Queries”. In: *Proc. ACM Symposium on Principles of Database Systems*. 2005, pp. 1–12 (cit. on p. 64).

[85] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86 (cit. on p. 9).

[86] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. “Type Casting Verification: Stopping an Emerging Attack Vector”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 81–96 (cit. on p. 36).

[87] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. “CP-Miner: Finding copy-paste and related bugs in large-scale software code”. In: *IEEE Transactions on software Engineering* 32.3 (2006), pp. 176–192 (cit. on p. 82).

[88] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. “Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution.” In: *Proc. Symposium on Network and Distributed System Security (NDSS)*. 2008, pp. 1–15 (cit. on p. 63).

[89] V Benjamin Livshits and Monica S Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis.” In: *Usenix Security*. 2005, pp. 18–18 (cit. on pp. 2, 7, 8, 44, 45).

[90] LLVM Compiler Infrastructure. *Clang Static Analyzer*. `http://clang-analyzer.llvm.org/`. Accessed: 03/23/17 (cit. on pp. 43, 56, 76).

[91] LLVM Compiler Infrastructure. *libFuzzer: a library for coverage-guided fuzz testing*. `http://llvm.org/docs/LibFuzzer.html`. Accessed: 03/23/17 (cit. on p. 57).

[92] Tony Long. *Feb. 7, 2000: Mafiaboy’s Moment*. `https://www.wired.com/2007/02/feb-7-2000-mafiaboys-moment-2/` (cit. on p. 17).

[93] ZDNet Magazine. *CES 2009: Palm announces the Palm Web OS and the Palm Pre device*. `http://www.zdnet.com/article/ces-2009-palm-announces-the-palm-web-os-and-the-palm-pre-device/` (cit. on p. 15).

[94] Andrian Marcus and Jonathan I Maletic. “Identification of high-level concept clones in source code”. In: *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. IEEE. 2001, pp. 107–114 (cit. on p. 82).

[95] John Markoff. *Computer intruder is put on probation and fined $10,000*. `https://www.nytimes.com/1990/05/05/us/computer-intruder-is-put-on-probation-and-fined-10000.html?sq=robert+tappan+morris&scp=2&st=nyt`. 1990 (cit. on p. 17).

[96] MITRE.org. *CVE-2014-0160: The Heartbleed Bug*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`. Accessed: 03/23/17 (cit. on p. 58).

[97] 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 (cit. on p. 58).

[98] 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 (cit. on p. 58).

[99] MITRE.org. *CVE-2017-0144: Remote code execution in SMBv1 server*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8317`. Accessed: 03/23/17 (cit. on p. 1).

[100] David Molnar, Xue Cong Li, and David Wagner. “Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs.” In: *Proc. Usenix Security Symp.* Vol. 9. 2009, pp. 67–82 (cit. on p. 59).

[101] M Mongiovi, G Giannone, A Fornaia, G Pappalardo, and E Tramontana. “Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in Java applications”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM. 2015, pp. 1573–1579 (cit. on p. 2).

[102] Mozilla Foundation. *How do I tell if my connection to a website is secure?* `https://support.mozilla.org/en-US/kb/how-do-i-tell-if-my-connection-is-secure` (cit. on p. 21).

[103] Mozilla Foundation. *Web application specification*. `http://mozilla.github.io/webapps-spec/`. Accessed: 2014-03-08 (cit. on p. 23).

[104] Mozilla Inc. *Firefox OS Simulator*. `https://developer.mozilla.org/en-US/docs/Archive/B2G_OS/Simulator` (cit. on p. 18).

[105] Mozilla Inc. *ScanJS*. `https://github.com/mozilla/scanjs`. Accessed: 2014-03-10 (cit. on pp. 15, 17).

[106] MozillaWiki. *App Permissions*. `https://developer.mozilla.org/en-US/Apps/Developing/App_permissions`. Accessed: 2014-03-10 (cit. on p. 24).

[107] *nDPI: Open and Extensible LGPLv3 Deep Packet Inspection Library*. `http://www.ntop.org/products/deep-packet-inspection/ndpi/`. Accessed: 03/23/17 (cit. on p. 47).

[108] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100 (cit. on pp. 2, 43).

[109] *Network Security Services*. `https://developer.mozilla.org/en/docs/NSS`. Accessed: 2014-03-08 (cit. on p. 19).

[110] IETF Network Working Group. *MPLS Label Stack Encoding*. `https://tools.ietf.org/html/rfc3032`. Accessed: 01-06-2016 (cit. on p. 3).

[111] Network Working Group, Internet Engineering Task Force. *Internet Security Glossary, Version 2*. `https://tools.ietf.org/html/rfc4949` (cit. on p. 2).

[112] James Newsome and Dawn Song. “Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software”. In: *In In Proceedings of the 12th Network and Distributed Systems Security Symposium*. Internet Society. 2005 (cit. on p. 5).

[113] NIST. *SAMATE - Software Assurance Metrics And Tool Evaluation*. `http://samate.nist.gov/Main_Page.html` (cit. on p. 35).

[114] NIST. *Test Suites, Software Assurance Reference Dataset*. `http://samate.nist.gov/SRD/testsuite.php` (cit. on p. 35).

[115] Hilarie Orman. “The Morris worm: A fifteen-year perspective”. In: *IEEE Security & Privacy* 99.5 (2003), pp. 35–43 (cit. on p. 1).

[116] PCWorld Magazine. *First Look at Mozilla’s Web Platform for Phones: ’Boot to Gecko’*. `https://www.pcworld.com/article/250879/first_look_at_mozilla_s_web_platform_for_phones_boot_to_gecko.html` (cit. on p. 15).

[117] *Peach Fuzzer*. `http://www.peachfuzzer.com/`. Accessed: 03/23/17 (cit. on pp. 2, 59).

[118] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. “Detection of recurring software vulnerabilities”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM. 2010, pp. 447–456 (cit. on p. 82).

[119] *PHP Bug Bounty Program*. `https://hackerone.com/php` (cit. on p. 37).

[120] *PHP::Sec Bug, 67492*. `https://bugs.php.net/bug.php?id=67492` (cit. on p. 38).

[121] *PHP::Sec Bug, 69085*. `https://bugs.php.net/bug.php?id=69085` (cit. on p. 38).

[122] *PHP::Sec Bug, 69152*. `https://bugs.php.net/bug.php?id=69152` (cit. on p. 38).

[123] *pinctrl: Be more careful in parsing DHCPv6 and DNS*. `https://mail.openvswitch.org/pipermail/ovs-dev/2017-May/332712`. Accessed: 24/5/2017 (cit. on pp. 74, 80).

[124] Mozilla Press. *Firefox OS Unleashes the Future of Mobile*. `https://blog.mozilla.org/press/2014/02/firefox-os-future-2/` (cit. on pp. 15, 17).

[125] Mohit Rajpal, William Blum, and Rishabh Singh. “Not all bytes are equal: Neural byte sieve for fuzzing”. In: *CoRR* abs/1711.04596 (2017). arXiv: `1711.04596` (cit. on p. 88).

[126] David A. Ramos and Dawson Engler. “Under-Constrained Symbolic Execution: Correctness Checking for Real Code”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64 (cit. on p. 44).

[127] *Report 73245: Type-confusion Vulnerability in SoapClient*. `https://hackerone.com/reports/73245` (cit. on p. 38).

[128] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise interprocedural dataflow analysis via graph reachability”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1995, pp. 49–61 (cit. on p. 45).

[129] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise interprocedural dataflow analysis via graph reachability”. In: *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1995, pp. 49–61 (cit. on p. 54).

[130] Henry Gordon Rice. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366 (cit. on pp. 5, 48).

[131] *Scan-build*. `http://clang-analyzer.llvm.org/scan-build.html` (cit. on p. 31).

[132] R. Scandariato, J. Walden, and W. Joosen. “Static analysis versus penetration testing: A controlled experiment”. In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2013, pp. 451–460 (cit. on p. 27).

[133] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)”. In: *Proc. IEEE Security & Privacy*. 2010, pp. 317–331 (cit. on p. 8).

[134] Jaume Segarra. *M2S: Free SMS*. `https://sourceforge.net/projects/m2s-free-sms/` (cit. on p. 24).

[135] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Proc. USENIX Annual Technical Conference (ATC)*. 2012, pp. 28–28 (cit. on pp. 2, 5, 70, 71).

[136] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC’12. Boston, MA: USENIX Association, 2012, pp. 28–28 (cit. on pp. 43, 60).

[137] Bhargava Shastry. *afl-sancov*. `https://github.com/bshastry/afl-sancov` (cit. on p. 11).

[138] Bhargava Shastry. *Mélange checkers*. `https://github.com/bshastry/melange-checkers` (cit. on p. 11).

[139] Bhargava Shastry and Markus Leutner. *Orthrus: A tool to manage, conduct and assess dictionary-based fuzz testing*. `https://github.com/test-pipeline/orthrus` (cit. on p. 11).

[140] Bhargava Shastry, Markus Leutner, Tobias Fiebig, et al. “Static Program Analysis as a Fuzzing Aid”. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis. Cham: Springer International Publishing, 2017, pp. 26–47 (cit. on p. ix).

[141] Bhargava Shastry, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. “Towards Vulnerability Discovery Using Staged Program Analysis”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 78–97 (cit. on p. ix).

[142] Dawn Song, David Brumley, Heng Yin, et al. “BitBlaze: A New Approach to Computer Security via Binary Analysis”. In: *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.* Hyderabad, India, Dec. 2008 (cit. on p. 2).

[143] *Source Code Analysis for Security through LLVM*. `http://llvm.org/devmtg/2014-10/Slides/Zhao-SourceCodeAnalysisforSecurity.pdf` (cit. on pp. 44, 45).

[144] “Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing”. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, 2017 (cit. on p. ix).

[145] Nick Stephens, John Grosen, Christopher Salls, et al. “Driller: Augmenting fuzzing through selective symbolic execution”. In: *Proc. Symposium on Network and Distributed System Security (NDSS)*. 2016 (cit. on p. 83).

[146] Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, et al. “Taking Control of SDN-based Cloud Systems via the Data Plane”. In: *Proceedings of the Symposium on SDN Research*. SOSR ’18. Los Angeles, CA, USA: ACM, 2018, 1:1–1:15 (cit. on pp. x, 4).

[147] Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, et al. “The vAMP Attack: Taking Control of Cloud Systems via the Unified Packet Parser”. In: *Proceedings of the 2017 on Cloud Computing Security Workshop*. CCSW ’17. Dallas, Texas, USA: ACM, 2017, pp. 11–15 (cit. on p. x).

[148] *ThreadSanitizer, MemorySanitizer, Scalable Run-time Detection of Uninitialized Memory Reads and Data Races with LLVM Instrumentation*. `http://llvm.org/devmtg/2012-11/Serebryany_TSan-MSan.pdf` (cit. on pp. 38, 43).

[149] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. “Seven pernicious kingdoms: A taxonomy of software security errors”. In: *Security & Privacy, IEEE* 3.6 (2005), pp. 81–84 (cit. on p. 36).

[150] The European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council*. `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2016.119.01.0001.01.ENG&toc=OJ:L:2016:119:TOC` (cit. on p. 87).

[151] Tommi Unruh, Bhargava Shastry, Malte Skoruppa, et al. “Leveraging Flawed Tutorials for Seeding Large-Scale Web Vulnerability Discovery”. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, 2017 (cit. on p. x).

[152] J. Viega, J.T. Bloch, Y. Kohno, and Gary McGraw. “ITS4: a static vulnerability scanner for C and C++ code”. In: *Computer Security Applications, 2000. ACSAC ’00. 16th Annual Conference*. Dec. 2000, pp. 257–267 (cit. on pp. 2, 7).

[153] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, et al. “Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.” In: *NDSS*. Vol. 2007. 2007, p. 12 (cit. on p. 2).

[154] Georgia Weidman. *Penetration testing: a hands-on introduction to hacking*. No Starch Press, 2014 (cit. on pp. 15, 18).

[155] Norman Wilde and Michael C Scully. “Software reconnaissance: Mapping program features to code”. In: *Journal of Software: Evolution and Process* 7.1 (1995), pp. 49–62 (cit. on p. 5).

[156] Wilkerson, Daniel. *CQUAL++*. `https://daniel-wilkerson.appspot.com/oink/qual.html`. Accessed: 2015-03-26 (cit. on pp. 43, 44).

[157] *WLLVM: Whole-program LLVM*. `https://github.com/travitch/whole-program-llvm` (cit. on p. 31).

[158] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. “Automatic Network Protocol Analysis”. In: *Proc. Symposium on Network and Distributed System Security (NDSS)*. 2008 (cit. on p. 63).

[159] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. “Automatic inference of search patterns for taint-style vulnerabilities”. In: *Proc. IEEE Security & Privacy*. 2015, pp. 797–812 (cit. on pp. 2, 65, 83, 88).

[160] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. “Generalized vulnerability extrapolation using abstract syntax trees”. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM. 2012, pp. 359–368 (cit. on pp. 7, 43, 44, 88).

[161] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. “Generalized vulnerability extrapolation using abstract syntax trees”. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM. 2012, pp. 359–368 (cit. on p. 82).

[162] Michal Zalewski. *afl-fuzz: making up grammar with a dictionary in hand*. `https://lcamtuf.blogspot.de/2015/01/afl-fuzz-making-up-grammar-with.html`. Accessed: 03/23/17. 2015 (cit. on p. 50).

[163] Michal Zalewski. *american fuzzy lop*. `http://lcamtuf.coredump.cx/afl/`. Accessed: 03/23/17 (cit. on p. 2).