Grenoble INP | **UGA**
Master CybserSecurity | **Year 2019-20**

Software Security & Secure Programming (L. Mounier)

Written Exam - Tuesday January the 21st, 2019

Duration: 2 hours – Answers can be written either in English or in French.
All documents allowed apart books – Calculators are forbidden.

This exam contains two distinct parts:

1. One exercise, supposed to be solved in about 1h;
2. Some questions on a research paper, allow 1h to read the paper and answer these questions.

Exercise. (∼ 10 pts)

We consider the following C program:

```
 1 int main() {
 2 unsigned char x, y, i ;
 3 char T[10] ; // T indices are in [0,9]
 4
 5 x=read(); // x takes a value in [0,255]
 6
 7 if (x+3 < 12)
 8       T[x]=42 ;
 9
10 y=x+1 ;
11 for (i=y ; i<3; i++)
12     y = y+2 ;
13
14 T[y]=12 ;
15 return 0 ;
16 }
```

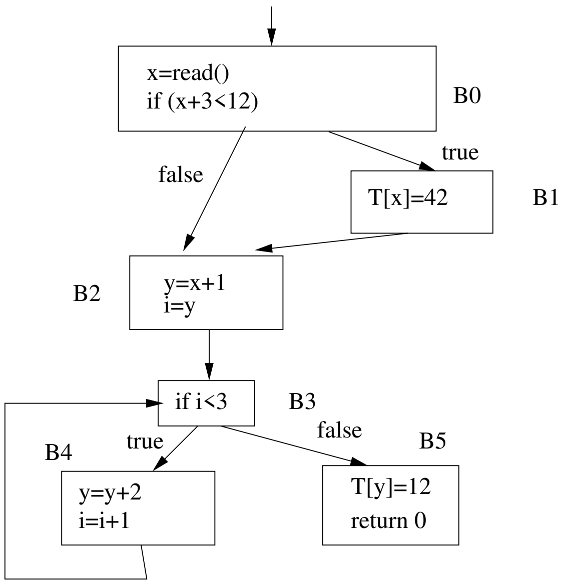


Figure 1: A vulnerable program and its Control-Flow Graph (CFG)

This program contains (at least !) two possible *buffer-overflow vulnerabilities* at lines 8 and 14. Note that **unsigned chars** values are in $[0, 255]$, with **wrap-around** in case of overflow.

In the following we investigate how much code analysis techniques would be able to detect if these vulnerabilities could be triggered at runtime.

We first consider a static verification approach, based on **abstract interpretation**, for instance with a tool like Frama-C “value analysis”. We use the *interval abstract domain*.

Q1. Give the relevant results produced by an automated *value analysis* on the code of Figure 1, namely the set of intervals obtained for variables `x`, `y` and `i` at lines 8 and 14 (using interval propagation rules). You can use widening/narrowing operators if you want (but it is not mandatory on this example).

Q2. Using the results of **Q1**, what can you conclude about the vulnerabilities of lines 8 and 14 ?

Q3. When running the Frama-C value-analysis on this example we get the following results:

- no runtime error at line 8 (`x` value is **proved** to be in $[0, 9]$);
- possible runtime error at line 14 (no **guarantee** that `x` value is in $[0, 9]$).

How do you explain these results ? Are they false positives ? What can we conclude at this stage regarding vulnerabilities of lines 8 and 14 ?

We now run a **symbolic execution** on this same program, for instance with a tool like PathCrawler.

Q4. We consider the (single) CFG execution path B0-B1 leading to line 8.

1. Give its path predicate (on variables `x`, `y` and `i`).
2. Give a solution (i.e., an input value of `x`) for this path predicate (making this path **executable**).
3. Give a necessary condition to follow this path **and activate the vulnerability**.
   Is there a solution (i.e., a concrete input value of `x`) which satisfies this condition ?

Q5. We now consider CFG execution paths leading to line 14.

1. how many such paths can we enumerate ?
2. We focus on the path B0-B1-B2-B3-B4-B3-B4-B3-B4-B3-B5:
   (a) Give its path predicate (on variables `x`, `y` and `i`).
   (b) Give a necessary condition to follow this path and activate the vulnerability. Is there a solution (i.e., an input value of `x`) which satisfies this condition ?

What can we conclude at this stage regarding vulnerabilities of lines 8 and 14 ?

Q6. We now suppose that line 11 is replaced by the following one:

```
for (i=0 ; i<y; i++)
```

Explain (informally, without running the example by hand again) what would be the impact of this modification on the results produced by each tool (Frama-C and PathCrawler) ?

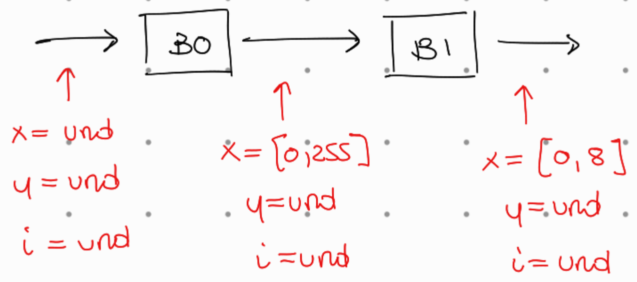
Questions on a research paper. (∼ 10 pts)

Read the paper given in appendix, answering the following questions as long as you read it. The objective of this part of the exam is to evaluate your ability to understand a description of a vulnerability detection tool (its strengths and limitations, with respect to other approaches you know). When answering the following questions you should not copy entire sentences from the paper but rather illustrate your point with examples and comments from your own.

Q1

The two lines that have been considered are line 8 and line 14.
They both contain an access to mem $\rightarrow$ $T[x] = 42$ and $T[y] = 12$

- at line 8 we are in B1, so we followed the path $B0 \rightarrow B1$



- at line 14 we are in B5 and we may have followed different paths.

  x is relevant only to the assignment of y so we may consider the worst case scenareo : $x = [0, 255]$

  At B2 $\rightarrow$ $y = [0,255]$ since we have wrap-arround
  $\rightarrow$ $i = [0,255]$

  At B3 $\rightarrow$ $y = [0,255]$ and $i = [0,255]$

Q2

Using the previos results we can conclude that the vuln at line 8 will not be activated, while at line 14 we may encounter a buffer overflow.

Q3

By the output of frama-c we are sure that the vuln at line 8 will not be activated and that we have no guarantee that the vuln at line 14 won't be executed.
The main difference is made by the condition at line 7. "if(x+3<12)"

Since frama-c is able to evaluate that and exclude the possibility of overflow.

Remark:

Frama-c may produce false positive but no false negative since it is supposed to overapproximate the values in the VSA.

Q4

Path B0 → B1

Predicate on x, y, i : ($x_0 < 9$ and $x_0 \geq 0$) or ($x_0 \geq 253$ or $x \leq 255$)

↑ This is because we have wrap around

Solution : $x_0 = 0$ is a solution

Activate vuln : ($x_0 \geq 253$ or $x \leq 255$) or ($x_0 \geq 9$ and $x_0 \geq 0$) → To reach B1

and ($x_0 \leq 0$ or $x > 8$) → To trigger the vuln

$x = 253$ will activate the vuln

Q5

2 simple path to B5 (avoiding the loop)

+ 6 path entering the loop (3 from B2 and 3 from B1)

8 path in total

Now considering the path : B0 − B1 − B2 − B3 − B4 − B3 − B4 − B3 − B4 − B3 − B5

Path predicate :

$x_0 + 3 < 12$ and $y_0 = x_0 + 1$ and $i_0 = y_0$ and

$i_0 < 3$ and $y_1 = y_0 + 2$ and $i_1 = i_0 + 1$

$i_1 < 3$ and $y_2 = y_1 + 2$ and $i_2 = i_1 + 1$

$i_2 < 3$ and $y_3 = y_2 + 2$ and $i_3 = i_2 + 1$

$i_3 \geq 3$ and ($y_3 < 0$ or $y_3 > 8$

No solution for this path predicate

Q6

Frama-c will not be able to keep track of the ralation $i<y$ thus leading to many new potential eror allarms.
Path Crawler may no longer be able to explore all paths since the number increased significantly.

Q1 [section 1].

1. What are the main objectives of the work described in this paper ?
2. What is claimed to be original in the proposed approach ?

Q2 [section 2].

1. Indicate what are the advantages of a *context-sensitive* and *path-sensitive* static analysis. You can use a small code example to illustrate your point.
2. According to you, is the Frama-C "value analysis" (used in the labs) path-sensitive ?

Q3 [section 3].

1. On of the defect targeted by Mélange is *uninitialized reads*, as illustrated in Listing 1.1. Explain why it is a security issue, and what could be the potential gain for an attacker.
2. What are the difficulties (explained in section 3.3) when building a call graph[1] from an object oriented program ?

[section 4.1 and 4.2 can be skipped]

Q4 [section 4].

1. Explain why *type confusion* (as explained in section 4.3) is a security issue, giving a concrete example.
2. Why is it useful to "match Mélange findings" with MemorySanitizer ?
3. Explain why Mélange can provide *false positives*, and why it could be a problem for the users.

Q5 [sections 5 and 6]

1. As a potential user, how would you compare Mélange with the other kinds of tools seen during the course, namely AFL, PathCrawler and the value analysis of Frama-C ? (with respect to each of these 3 tools, tell what are the benefits/drawbacks of Mélange).
2. The paper says that path sensitivity is implemented using symbolic execution. What main limitations do you see with respect to this technique ?
3. The paper never mentions *false negatives* (i.e., real vulnerabilities missed by the tool). Is it because Mélange is not supposed to produce false negatives ?
4. Mélange operates at the LLVM level, which is a byte-code used as an intermediate compilation step. Do you think the same approach could be implemented directly on binary code (why, or why not) ? And, in practice, what about the relevance of detecting vulnerabilities on LLVM code instead of binary code ?

[1]the call graph indicates which methods are called by which other methods

Towards Vulnerability Discovery Using Staged Program Analysis

Bhargava Shastry[1], Fabian Yamaguchi[2], Konrad Rieck[2], and Jean-Pierre Seifert[1]

[1] Security in Telecommunications, TU Berlin, Germany
[2] Institute of System Security, TU Braunschweig, Germany

Abstract. Eliminating vulnerabilities from low-level code is vital for securing software. Static analysis is a promising approach for discovering vulnerabilities since it can provide developers early feedback on the code they write. But, it presents multiple challenges not the least of which is understanding what makes a bug exploitable and conveying this information to the developer. In this paper, we present the design and implementation of a practical vulnerability assessment framework, called *Mélange*. Mélange performs data and control flow analysis to diagnose potential security bugs, and outputs well-formatted bug reports that help developers *understand* and *fix* security bugs. Based on the intuition that real-world vulnerabilities manifest themselves across multiple parts of a program, Mélange performs both local and global analyses. To scale up to large programs, global analysis is demand-driven. Our prototype detects multiple vulnerability classes in *C* and *C++* code including type confusion, and garbage memory reads. We have evaluated Mélange extensively. Our case studies show that Mélange scales up to large codebases such as Chromium, is easy-to-use, and most importantly, capable of discovering vulnerabilities in real-world code. Our findings indicate that static analysis is a viable reinforcement to the software testing tool set.

Keywords: Program Analysis, Vulnerability Assessment, LLVM

1 Introduction

Vulnerabilities in popularly used software are not only detrimental to end-user security but can also be hard to identify and fix. Today's highly inter-connected systems have escalated the damage inflicted upon users due to security compromises as well as the cost of fixing vulnerabilities. To address the threat landscape, software vendors have established mechanisms for software quality assurance and testing. A prevailing thought is that security bugs identified and fixed early impose lower costs than those identified during the testing phase or in the wild. Thus, vulnerability re-mediation—the process of identifying and fixing vulnerabilities—is being seen as part of the software development process rather than in isolation [28].

Program analysis provides a practical means to discover security bugs during software development. Prior approaches to vulnerability discovery using static code analysis have ranged from simple pattern-matching to context and path-sensitive data-flow analysis. For instance, ITS4 [42]—a vulnerability scanner for C/C++ programs—parses source code and looks up lexical tokens of interest against an existing vulnerability database. In our initial experiments, the pattern-matching approach employed by ITS4 produced a large number of warnings against modern C, and C++ codebases. On the contrary, security vulnerabilities are most often, subtle corner cases, and thus rare. The approach taken by ITS4 is well-suited for extremely fast analysis, but the high amount of manual effort required to validate warnings undermines the value of the tool itself.

On the other end of the spectrum, the Clang Static Analyzer [4] presents an analytically superior approach for defect discovery. Precise—context and path sensitive—analysis enables Clang SA to warn only when there is evidence of a bug in a feasible program path. While precise warnings reduce the burden of manual validation, we find that Clang SA's local inter-procedural analysis misses security bugs that span file boundaries. The omission of bugs that span file boundaries is significant especially for object-oriented code[3], where object implementation and object use are typically in different source files. A natural solution is to make analysis global. However, global analysis does not scale up to large programs.

In this paper, we find a middle ground. We present the design and implementation of Mélange, a vulnerability assessment tool for C and C++ programs, that performs both local and global analysis in stages to discover potential vulnerabilities spanning source files. Mélange has been implemented as an extension to the LLVM compiler infrastructure [32]. To keep analysis scalable, Mélange performs computationally expensive analyses locally (within a source file), while performing cheaper analyses globally (across the whole program). In addition, global analysis is demand-driven: It is performed to validate the outcome of local analyses. To provide good diagnostics, Mélange primarily analyzes source code. It outputs developer-friendly bug reports that point out the *exact* position in source code where potential vulnerabilities exist, why they are problematic, and how they can be remedied.

Results from our case studies validate our design decisions. We find that Mélange is capable of highlighting a handful of problematic corner cases, while scaling up to large programs like Chromium, and Firefox. Since Mélange is implemented as an extension to a widely used compiler toolchain (Clang/LLVM), it can be invoked as part of the build process. Moreover, our current implementation is fast enough to be incorporated into nightly builds[4] of two large codebases (MySQL, Chromium), and with further optimizations on the third (Firefox). In summary, we make the following contributions.

[3] All the major browsers including Chromium and Firefox are implemented in object-oriented code.
[4] Regular builds automatically initiated overnight on virtual machine clusters

1. We present the design and implementation of Mélange, an extensible program analysis framework.
2. We demonstrate the utility of Mélange by employing it to detect multiple classes of vulnerabilities, including garbage reads and incorrectly typed data, that are known to be a common source of exploitable vulnerabilities.
3. We evaluate Mélange extensively. We benchmark Mélange against NIST's Juliet benchmark [36] for program analysis tools. Mélange has thus far detected multiple known vulnerabilities in the PHP interpreter, and Chromium codebases, and discovered a new defect in Firefox.

2 Background: Clang and LLVM

Mélange is anchored in the Clang/LLVM open-source compiler toolchain [13], an outcome of pioneering work by Lattner et al. [32]. In this section, we review components of this toolchain that are at the core of Mélange's design. While Clang/LLVM is a compiler at heart, it's utility is not limited to code generation/optimization. Different parts of the compiler front-end (Clang) and back-end (LLVM) are encapsulated into libraries that can be selectively used by client systems depending on their needs. Thus, the LLVM project lends itself well to multiple compiler-technology-driven use-cases, program analysis being one of them.

We build Mélange on top of the analysis infrastructure available within the LLVM project. This infrastructure mainly comprises the Clang Static Analyzer—a source code analyzer for *C*, C++, and *Objective-C* programs—and the LLVM analyzer/optimizer framework which permits analysis of LLVM Bitcode. In the following paragraphs, we describe each of these components briefly.

2.1 Clang Static Analyzer

The Clang Static Analyzer (Clang SA) is similar in spirit to *Metal/xgcc*, which its authors classify as a "Meta-level Compilation" (MC) framework [21,24]. The goal of an MC framework is to allow for modular extensions to the compiler that enable checking of domain-specific program properties. Abstractly viewed, an MC framework comprises a set of *checkers* (domain-specific analysis procedures) and a compilation system.

The division of labor envisioned by Hallem et al. [24] is that *checkers* only *encode* the property to check, leaving the mechanics of the actual checking to the compilation system. The compilation system facilitates checking by providing the necessary analysis infrastructure. Figure 1 shows how an MC framework is realized in Clang SA. Source files are parsed and subsequently passed on to the Data-Flow Analysis engine (*DFA* engine), which provides the analysis infrastructure required by checkers. Checkers encode the program property to be checked and produce bug reports if a violation is found. Bug reports are then reviewed by a human analyst.

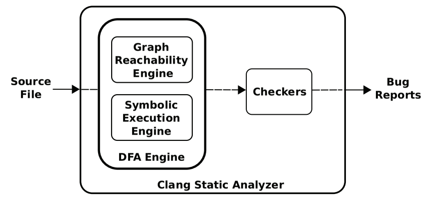


Fig. 1: Clang Static Analyzer overview

Data-flow Analysis Engine Clang SA performs *Context* and *Path* sensitive inter-procedural data-flow analysis. Context sensitivity means that the analysis preserves the calling context of function calls; path sensitivity means that the analysis explores paths forked by branch statements *independently*. Context sensitivity is realized in the *Graph Reachability Engine* which implements a namesake algorithm proposed by Reps et al. [37]. Path sensitivity is implemented in the *Symbolic Execution Engine*. The symbolic execution engine uses static Forward Symbolic Execution (FSE) [38] to explore program paths in a source file.

Checkers Checkers implement domain-specific checks and issue bug reports. Clang SA contains a default suite of checkers that implement a variety of checks including unsafe API usage, and memory access errors. More importantly, the checker framework in Clang SA can be used by programmers to add custom checks. To facilitate customized checks, Clang SA exposes callbacks (as APIs) that *hook* into the DFA engine at pre-defined program locations. Clang SA and its checkers seen together, demonstrate the utility of meta-level compilation.

2.2 LLVM Pass Infrastructure

The LLVM pass infrastructure [13] provides a modular means to perform analyses and optimizations on an LLVM Intermediate Representation (IR) of a program. LLVM IR is a typed, yet source-language independent representation of a program that facilitates *uniform* analysis of whole-programs or whole-libraries.

Simply put, an LLVM Pass is an operation (procedure invocation) on a unit of LLVM IR code. The granularity of code operated on can vary from a *Function* to an entire program (*Module* in LLVM parlance). Passes may be run in sequence, allowing a successive *pass* to reuse information from (or work on a transformation carried out by) preceding *passes*. The LLVM pass framework provides APIs to tap into source-level meta-data in LLVM IR. This provides a means to bridge the syntactic gap between source-level and IR-level analyses. Source literals may be matched against LLVM IR meta-data programmatically. Mélange takes this approach to teach the LLVM pass what a source-level bug report means.

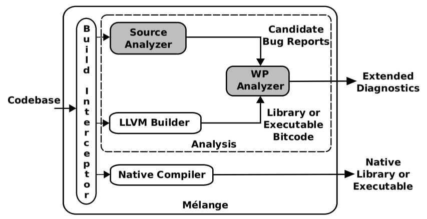


Fig. 2: Mélange overview

3 Mélange

Our primary goal is to develop an early warning system for security-critical software defects. We envision Mélange as a tool that assists a developer in identifying, and fixing potential security bugs *during* software development. Figure 2 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.

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. (§3.1)
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. (§3.1)
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)
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. (§3.3)

3.1 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* [12], 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) [14], 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 Source Analyzer

The source analyzer assists Mélange in searching for potential bugs in source code. We build an *event collection* system to aid this search. Our event collection system is flexible enough to cater to multiple use-cases. It can be employed to detect both traditional taint-style vulnerabilities as well as semantic defects. The 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.

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[5] of C++ objects, incorrect type casts in PHP interpreter codebase, and other Common Weakness Enumerations (see §4).

[5]The algorithm for flagging garbage reads is based on a variation of gen-kill sets [30].

We demonstrate the utility of the proposed system by using the code snippet shown in Listing 1.1 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, $\mathit{\Delta}_F$ 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.

```
 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 1.1: 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.

In Listing 1.1, when function `foo::isZero` in file `foo.cpp` is being analyzed, the checker adds class member `foo::x` to the UseWithoutDef set of $\mathit{\Delta}_{foo::isZero}$ 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 $\mathit{\Delta}_{foo::foo}$ is empty. Finally, the checker compares set memberships across analysis contexts. Since `foo::x` is marked as a use without a valid definition in $\mathit{\Delta}_{foo::isZero}$ and `foo::x` is not a member of the Def set in the constructor function's analysis summary ($\mathit{\Delta}_{foo::foo}$), 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 1.2 shows how candidate bug reports are encoded. The 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 paragraph, we describe how candidate bug reports are analyzed by our whole-program analyzer.

3.3 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 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) [20]. 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 domain-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 1.1), loads and stores to the `foo::x` class member indicated in candidate bug report (Listing 1.2) 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 1.2).

```
// 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 1.2: Candidate bug report (top) and whole-program bug report (bottom) for garbage read in the running example shown in Listing 1.1.

4 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? (§4.1)
– How does Mélange perform against static analysis benchmarks? (§4.2)
– How does Mélange fare against known security vulnerabilities? (§4.3)
– What is the analysis run-time and effectiveness of Mélange against large well-tested codebases? (§4.4)

4.1 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 §4.4). 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.

4.2 NIST Benchmarks

We used static analysis benchmarks released under NIST's SAMATE project [35] for benchmarking Mélange's detection rates. In particular, the Juliet C/C++ test suite (version 1.2) [36] was used to measure true and false positive detection rates for defects spread across multiple categories. The Juliet suite comprises test sets for multiple defect types. Each test set contains test cases for a specific Common Weakness Enumeration (CWE) [41]. 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 [33] for exploiting popular applications.

Figure 3 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 4). 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.

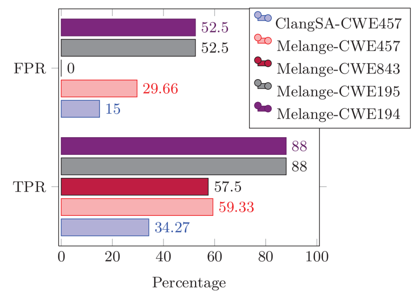


Fig. 3: Juliet test suite: True Positive Rate (TPR) and False Positive Rate (FPR) for Mélange, and Clang Static Analyzer. Clang SA supports CWE457 only.

4.3 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 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 [7].

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.

4.4 Case Studies

To further investigate the practical utility of Mélange, we conducted case studies with three popular open-source projects, namely, Chromium, Firefox, and

Codebase	CVE ID (Rating)	Bug ID	Vulnerability	Known/New
PHP	CVE-2015-4147	69085 [9]	Type-confusion	Known
PHP	CVE-2015-4148	69085 [9]	Type-confusion	Known
PHP	CVE-2014-3515	67492 [8]	Type-confusion	Known
PHP	Unassigned	73245 [11]	Type-confusion	Known
PHP	Unassigned	69152 [10]	Type-confusion	Known
Chromium	(Medium-Severity)	411177 [2]	Garbage read	Known
Chromium	None	436035 [3]	Garbage read	Known
Firefox	None	1168091 [1]	Garbage read	New

Table 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.

MySQL. We focused on detecting garbage reads only. In the following paragraphs, we present results from our case studies emphasizing analysis effectiveness, and analysis run-time.

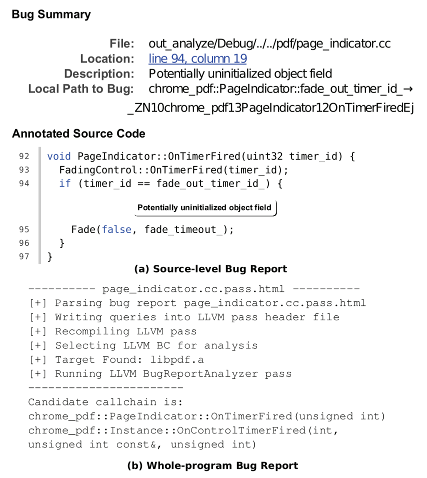
Software Versions: Evaluation was carried out for Chromium version 38 (dated August 2014), for Firefox revision 244208 (May 2015), and for MySQL version 5.7.7 (April 2015).

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.

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 1). Defects found by our prototype in MySQL codebase have been reported upstream and are being triaged. Figure 4 shows Mélange's bug report for a garbage read in the pdf library shipped with Chromium v38. The source-level bug report (Figure 4a) shows the line of code that was buggy. WP analyzer's bug report (Figure 4b) 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 [2, 3] generated by MemorySanitizer [40], a dynamic program analysis tool from Google. The new defect discovered in Firefox was reported upstream [1]. 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. 4: 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 1.3 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 1.3), 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.

Control-flow imprecision: Mélange's WP analyzer misses control-flow information at indirect call sites e.g., virtual function invocations. Thus, class

Codebase	Build Time	Analysis Run-time*				Bug Reports	
	N_t	SA_x	WPA_x	TA_x	$WPAvg_t$	Total	True Positives
Chromium	18m20s	29.09	15.49	44.58	7.5s	12	2
Firefox	41m25s	3.38	39.31	42.69	13m35s	16	1
MySQL	8m15s	9.26	21.24	30.50	2m26s	32	-

*All terms except $WPAvg$ are normalized to native compilation time

Table 2: Mélange: Analysis summary for large open-source projects. True positives for MySQL have been left out since we are awaiting confirmation from its developers.

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 [16] to improve Mélange's control-flow precision.

```
 1  AltSvcMapping::AltSvcMapping(...) {
 2    if (NS_FAILED(SchemeIsHTTPS(originScheme, mHttps))) {
 3      ...
 4    }
 5  }
 6  void AltSvcMapping::GetConnectionInfo(...) {
 7    // ci is an object on the stack
 8    ci->SetInsecureScheme(!mHttps);
 9    ...
10  }
11  static nsresult SchemeIsHTTPS(const nsACString &
        originScheme, bool &outIsHTTPS)
12  {
13    outIsHTTPS =
          originScheme.Equals(NS_LITERAL_CSTRING("https"));
14    ...
15  }
```

Listing 1.3: Code snippet involving an aliased definition that caused a false positive in Mélange.

The last two columns of Table 2 present a summary of Mélange's bug reports for Chromium, Firefox, and MySQL projects once both stages of analysis have been completed. We find that Mélange's two-stage analysis pipeline is very effective at filtering through a handful of bug reports that merit attention. For instance, Mélange's analyses output only twelve bug reports for Chromium, a codebase that spans over 14 million lines of code. 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

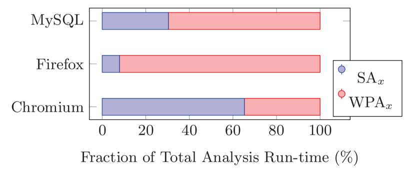


Fig. 5: For each codebase, its source and whole-program analysis run-times are shown as fractions (in %) of Mélange's total analysis run-time.

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 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 5 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.

4.5 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 [31]).

5 Related Work

Program analysis research has garnered attention since the late 70s. Lint [29], a C program checker developed at Bell Labs in 1977, was one of the first program analysis tools to be developed. Lint's primary goal was to check "portability, style, and efficiency" of programs. Ever since, the demands from a program checker have grown as new programming paradigms have been invented and programs have increased in complexity. This has contributed to the development of many commercial [5, 23, 27], closed-source [19], free [6], and open source [4, 15, 17, 18, 22, 26, 39, 40, 43, 44] tools. Broadly, these tools are based on *Model Checking* [17, 26], *Theorem Proving* [6], *Static Program Analysis* [4, 5, 19, 23, 27, 44], *Dynamic Analysis* [18, 34, 39, 40], or are hybrid systems such as AEG [15]. 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 fuzz testing (or fuzzing) instrumented programs. 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 [34]), or at compile time (as in AddressSanitizer or ASan [39]). Compile-time instrumentation has been preferred lately due to the poor performance of tools that employ run-time instrumentation.

While sanitizer tools such as ASan, and MemorySanitizer (MSan) are expected to have a zero false positive rate, practical difficulties, such as uninstrumented code in an external library, lead to false positives in practice. Thus, even run-time tools do not eliminate the need for manual validation of bug reports. To guarantee absence of uninitialized memory, MSan needs to monitor each and every load from/store to memory. This all-or-nothing philosophy poses yet another problem. Uninstrumented code in pre-compiled libraries (such as the C++ standard library) used by the program will invariably lead to false program crashes. Until these false crashes are rectified—either by instrumenting the code where the crash happens or by asking the tool to suppress the warning—the sanitizer tool is rendered unusable. Thus, use of MSan impinges on instrumentation of each and every line of code that is directly or indirectly executed by the program or maintenance of a blacklist file that records known false positives. Unlike MSan, not having access to library source code only lowers Mélange's analysis accuracy, but does not impede analysis itself. Having said that, Mélange will benefit from a mechanism to suppress known false positives. Overall, we believe that dynamic tools are invaluable for vulnerability assessment, and that a tool such as ours can complement them well.

Symbolic Execution Symbolic execution has been used to find bugs in programs, or to generate test cases with improved code coverage. KLEE [18], Clang SA [4], and AEG [15] use different flavors of forward symbolic execution for their own end. 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. Unlike KLEE and AEG, symbolic execution in Clang SA is done locally and hences scales up to large codebases. Anecdotal evidence suggests that KLEE and AEG don't scale up to large programs [25]. To the best of our knowledge, KLEE has not been evaluated against even medium-sized codebases let alone large codebases such as Firefox and Chromium.

Static Analysis Parfait [19] employs an analysis strategy that is similar in spirit to ours. It employs multiple stages of analysis, where each successive stage is more precise than the preceding stage. Parfait has been used for finding buffer overflows in C programs. In contrast, we have evaluated Mélange against multiple vulnerability classes. Mélange's effectiveness in detecting multiple CWEs validates the generality of its design. In addition, Mélange has fared well against multiple code paradigms: both legacy C programs and modern object-oriented code.

Like Yamaguchi et al. [44], our goal is to empower developers in finding multiple instances of a known defect. However, the approach we take is different. Yamaguchi et al. [44], use structural traits in a program's *AST* representation 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. CQUAL [22], and CQual++ [43], 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.

Finally, Clang Static Analyzer borrows ideas from several publications including (but not limited to) [24,37]. Inter-procedural context-sensitive analysis in Clang SA is based on the graph reachability algorithm proposed by Reps et al. [37]. Clang SA is also similar in spirit to Metal/xgcc [24].

6 Conclusion

We have developed Mélange, a static analysis tool for helping fix security-critical defects in open-source software. Our tool is premised on the intuition that vulnerability search necessitates multi-pronged analysis. We anchor Mélange in the Clang/LLVM compiler toolchain, leveraging source analysis to build a corpus of defects, and whole-program analysis to filter the corpus. We have shown that our approach is capable of identifying defects and vulnerabilities in open-source projects, the largest of which—Chromium—spans over 14 million lines of code. We have also demonstrated that Mélange's analyses are viable by empirically evaluating its run-time in an EC2 instance.

Since Mélange is easy to deploy in existing software development environments, programmers can receive early feedback on the code they write. Furthermore, our analysis framework is extensible via compiler plug-ins. This enables programmers to use Mélange to implement domain-specific security checks. Thus, Mélange complements traditional software testing tools such as fuzzers. Ultimately, our aim is to use the proposed system to help fix vulnerabilities in open-source software at an early stage.

Acknowledgments This work was supported by the following grants: 317888 (project NEMESYS), 10043385 (project Enzevalos), and RI 2468/1-1 (project DEVIL). Authors would like to thank colleagues at SecT and Daniel Defreez for valuable feedback on a draft of this paper, and Janis Danisevskis for discussions on the C++ standard and occasional code reviews.

References

1. Bugzilla@Mozilla, Bug 1168091. `https://bugzilla.mozilla.org/show_bug.cgi?id=1168091`
2. Chromium Issue Tracker, Issue 411177. `https://code.google.com/p/chromium/issues/detail?id=411177`
3. Chromium Issue Tracker, Issue 436035. `https://code.google.com/p/chromium/issues/detail?id=436035`
4. Clang Static Analyzer. `http://clang-analyzer.llvm.org/`, accessed: 2015-03-25
5. Coverity inc. `http://www.coverity.com/`

6. HAVOC. http://research.microsoft.com/en-us/projects/havoc/
7. PHP Bug Bounty Program. https://hackerone.com/php
8. PHP::Sec Bug, 67492. https://bugs.php.net/bug.php?id=67492
9. PHP::Sec Bug, 69085. https://bugs.php.net/bug.php?id=69085
10. PHP::Sec Bug, 69152. https://bugs.php.net/bug.php?id=69152
11. Report 73245: Type-confusion Vulnerability in SoapClient. https://hackerone.com/reports/73245
12. Scan-build. http://clang-analyzer.llvm.org/scan-build.html
13. The LLVM Compiler Infrastructure. http://llvm.org/
14. WLLVM: Whole-program LLVM. https://github.com/travitch/whole-program-llvm
15. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: Aeg: Automatic exploit generation. In: NDSS. vol. 11, pp. 59–66 (2011)
16. Bacon, D.F., Sweeney, P.F.: Fast static analysis of c++ virtual function calls. In: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 324–341. OOPSLA '96, ACM, New York, NY, USA (1996), http://doi.acm.org/10.1145/236337.236371
17. Ball, T., Rajamani, S.K.: The s lam project: debugging system software via static analysis. In: ACM SIGPLAN Notices. vol. 37, pp. 1–3. ACM (2002)
18. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
19. Cifuentes, C., Scholz, B.: Parfait: designing a scalable bug checker. In: Proceedings of the 2008 workshop on Static analysis. pp. 4–11. ACM (2008)
20. Dean, J., Grove, D., Chambers, C.: 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. pp. 77–101. Springer (1995)
21. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4. pp. 1–1. USENIX Association (2000)
22. Foster, J.S., Johnson, R., Kodumal, J., Terauchi, T., Shankar, U., Talwar, K., Wagner, D., Aiken, A., Elsman, M., Harrelson, C.: CQUAL: A tool for adding type qualifiers to C (2003), https://www.cs.umd.edu/~jfoster/cqual/, accessed: 2015-03-26
23. GrammaTech: CodeSonar. http://www.grammatech.com/codesonar
24. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for building system-specific, static analyses. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. pp. 69–82. PLDI '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/512529.512539
25. Heelan, S.: Vulnerability detection systems: Think cyborg, not robot. Security Privacy, IEEE 9(3), 74–77 (May 2011)
26. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. In: Model Checking Software, pp. 235–239. Springer (2003)
27. Hewlett Packard: Fortify Static Code Analyzer. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/
28. Howard, M., Lipner, S.: The security development lifecycle. O'Reilly Media, Incorporated (2009)

29. Johnson, S.: Lint, a C program checker. Bell Telephone Laboratories (1977)
30. Knoop, J., Steffen, B.: Efficient and optimal bit vector data flow analyses: a uniform interprocedural framework. Inst. für Informatik und Praktische Mathematik (1993)
31. Kremenek, T., Engler, D.: Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In: Proceedings of the 10th International Conference on Static Analysis. pp. 295–315. SAS'03, Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1760267.1760289
32. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization, 2004. CGO 2004. International Symposium on. pp. 75–86. IEEE (2004)
33. Lee, B., Song, C., Kim, T., Lee, W.: Type casting verification: Stopping an emerging attack vector. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 81–96. USENIX Association, Washington, D.C. (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee
34. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan notices. vol. 42, pp. 89–100. ACM (2007)
35. NIST: SAMATE - Software Assurance Metrics And Tool Evaluation. http://samate.nist.gov/Main_Page.html
36. NIST: Test Suites, Software Assurance Reference Dataset. http://samate.nist.gov/SRD/testsuite.php
37. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 49–61. ACM (1995)
38. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Security and Privacy (SP), 2010 IEEE Symposium on. pp. 317–331. IEEE (2010)
39. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference. pp. 28–28. USENIX ATC'12, USENIX Association, Berkeley, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2342821.2342849
40. Stepanov, E., Serebryany, K.: Memorysanitizer: fast detector of uninitialized memory use in c++. In: Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on. pp. 46–55. IEEE (2015)
41. Tsipenyuk, K., Chess, B., McGraw, G.: Seven pernicious kingdoms: A taxonomy of software security errors. Security & Privacy, IEEE 3(6), 81–84 (2005)
42. Viega, J., Bloch, J., Kohno, Y., McGraw, G.: Its4: a static vulnerability scanner for c and c++ code. In: Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference. pp. 257–267 (Dec 2000)
43. Wilkerson, Daniel: CQUAL++. https://daniel-wilkerson.appspot.com/oink/qual.html, accessed: 2015-03-26
44. Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference. pp. 359–368. ACM (2012)