2015 IEEE Symposium on Security and Privacy

Automatic Inference of Search Patterns for Taint-Style Vulnerabilities

Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck
University of Göttingen, Germany

***Abstract*—Taint-style vulnerabilities are a persistent problem in software development, as the recently discovered "Heartbleed" vulnerability strikingly illustrates. In this class of vulnerabilities, attacker-controlled data is passed unsanitized from an input source to a sensitive sink. While simple instances of this vulnerability class can be detected automatically, more subtle defects involving data flow across several functions or project-specific APIs are mainly discovered by manual auditing. Different techniques have been proposed to accelerate this process by searching for typical patterns of vulnerable code. However, all of these approaches require a security expert to manually model and specify appropriate patterns in practice.**

In this paper, we propose a method for automatically inferring search patterns for taint-style vulnerabilities in C code. Given a security-sensitive sink, such as a memory function, our method automatically identifies corresponding source-sink systems and constructs patterns that model the data flow and sanitization in these systems. The inferred patterns are expressed as traversals in a code property graph and enable efficiently searching for unsanitized data flows—across several functions as well as with project-specific APIs. We demonstrate the efficacy of this approach in different experiments with 5 open-source projects. The inferred search patterns reduce the amount of code to inspect for finding known vulnerabilities by 94.9% and also enable us to uncover 8 previously unknown vulnerabilities.

***Index Terms*—Vulnerabilities; Clustering; Graph Databases;**

I. Introduction

The discovery and elimination of vulnerabilities in software is a fundamental problem of computer security. Unfortunately, even subtle defects, such as a single missing authorization check or a slightly insufficient sanitization of data can already lead to severe security vulnerabilities in software. The necessity for development of more effective approaches for the discovery of such vulnerabilities has been made strikingly obvious by the recent "Heartbleed" vulnerability in the cryptographic library OpenSSL [1] and the "Shellshock" vulnerability in GNU Bash [2]. As programs are constantly modified and the properties of the platforms they operate on change, new vulnerabilities regularly emerge. In effect, vulnerability discovery becomes an on-going process, requiring experts with a deep understanding of the software in question and all the technologies its security relies upon.

Due to the diversity of vulnerable programming practices, security research has largely focused on detecting specific types of vulnerabilities. For example, fuzz testing [e.g., 20, 53] and symbolic execution [e.g., 49, 59] have been successfully applied to find memory corruption vulnerabilities, such as buffer overflows, integer overflows and format string vulnerabilities. In line with this research, a variety of approaches for detection of web application vulnerabilities have been proposed, for example for SQL injection flaws [e.g., 10, 26], cross-site scripting [e.g., 31, 48] and missing authorization checks [19, 51]. More recently, several researchers have recognized that many common vulnerabilities in both, system software and web applications, share an underlying theme rooted in information flow analysis: data propagates from an attacker-controlled input source to a sensitive sink without undergoing prior sanitization, a class of vulnerabilities referred to as *taint-style vulnerabilities* [see 9, 10, 26, 63].

Different approaches have been devised that enable mining for taint-style vulnerabilities using description languages that allow dangerous programming patterns to be precisely encoded [30, 35, 63]. In theory, this idea bares the possibility to construct a large database of patterns for known vulnerabilities that can be easily matched against source code. Unfortunately, similar to signature-based intrusion detection systems, constructing effective search patterns for vulnerabilities requires a security expert to invest a considerable amount of manual work. Starting from a security-sensitive sink, the expert needs to identify related input sources, data flows and corresponding sanitizations checks, which often involves a profound understanding of project-specific functions and interfaces.

In this paper, we present a method for automatically inferring search patterns for taint-style vulnerabilities from C source code. Given a sensitive sink, such as a memory or network function, our method automatically identifies corresponding source-sink systems in a code base, analyzes the data flow in these systems and generates search patterns that reflect the characteristics of taint-style vulnerabilities. To this end, we combine techniques from static program analysis and unsupervised machine learning that enable us to construct patterns that are usually identified by manual analysis and that allow for pinpointing insufficient sanitization, even if the data flow crosses several function boundaries and involves project-specific APIs. Analysts can employ this method to generate patterns for API functions known to commonly be associated with vulnerabilities, as well as to find instances of the same vulnerability spread throughout the code base.

We implement our approach by extending the analysis platform *Joern*[1] to support interprocedural analysis and developing a plugin for extracting and matching of search patterns, that is, robust descriptions of syntax, control flow and data flow that characterize a vulnerability. The platform is build on

[1] A Robust Code Analysis Platform for C/C++, http://mlsec.org/joern

DOI 10.1109/SP.2015.54

top of an efficient graph database and uses a so-called *code property graph* [63] for representing the syntax, control flow and data flow of source code. To leverage this representation, we express the inferred search patterns as graph traversals, which enables us to quickly pass through the graph and thereby scan large software projects for occurrences of potential vulnerabilities within a few minutes.

We empirically evaluate our method's ability to generate search patterns for known vulnerabilities with 5 open-source projects, showing that the amount of code to review can be reduced by 94.9%. Moreover, we demonstrate the practical merits of our method by discovering 8 previously unknown vulnerabilities using only a few generated search patterns.

In summary, our contributions are the following.

- *Extension of code property graphs.* We extend the recently presented code property graph [63] to include information about statement precedence and enable interprocedural analysis using graph database queries.
- *Extraction of invocation patterns.* We propose a novel method for extracting patterns of invocations from source code using clustering algorithms, including definitions of arguments and the sanitization they undergo.
- *Automatic inference of search patterns.* Finally, we show how invocation patterns can be translated into search patterns in the form of graph traversals that enable auditing large code bases.

The remainder of this paper is organized as follows: In Section II we present the basics of taint-style vulnerabilities and code property graphs. We then introduce our extension of code property graphs in Section III. Our method for automated inference of search patterns is presented in Section IV and evaluated in Section V. We discuss limitations of our approach and related work in Section VI and VII, respectively. Section VIII concludes the paper.

II. Background

Vulnerability discovery is a classic topic of computer security and consequently, many approaches have been presented, focusing on various types of vulnerabilities and technologies. In this section, we briefly review approaches related to our method. We begin by discussing the notion of *taint-style vulnerabilities* in Section II-A, as these are the types of vulnerabilities we deal with throughout the paper. We proceed to describe how these types of vulnerabilities can be discovered using *code property graphs* in Section II-B, a representation designed for pattern-based vulnerability discovery, which we extend for interprocedural analysis.

A. Taint-Style Vulnerabilities

The term *taint-style vulnerabilities* has its roots in taint analysis, a technique for tracing the propagation of data through a program. One goal of taint analysis is to identify data flows from attacker-controlled sources to security-sensitive sinks that do not undergo sanitization. This procedure requires the definition of (a) appropriate sources, (b) corresponding sinks and (c) sanitization rules. While at first, it may seem that only a handful of flaws can be described in this way, this vulnerability class fits many common security defects well, including different types of buffer overflows and other memory corruption flaws, SQL and command injection, as well as missing authorization checks.

```
1  /* ssl/d1_both.c */
2  // [...]
3  int dtls1_process_heartbeat(SSL *s)
4  {
5    unsigned char *p = &s->s3->rrec.data[0], *pl;
6    unsigned short hbtype;
7    unsigned int payload;
8    unsigned int padding = 16; /* Use minimum padding */
9    /* Read type and payload length first */
10   hbtype = *p++;
11   n2s(p, payload);
12   if (1 + 2 + payload + 16 > s->s3->rrec.length)
13     return 0; /* silently discard per RFC 6520 sec.4*/
14   pl = p;
15   // [...]
16   if (hbtype == TLS1_HB_REQUEST){
17    unsigned char *buffer, *bp;
18    int r;
19    // [...]
20    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
21    bp = buffer;
22    /* Enter response type, length and copy payload */
23    *bp++ = TLS1_HB_RESPONSE;
24    s2n(payload, bp);
25    memcpy(bp, pl, payload);
26    bp += payload;
27    /* Random padding */
28    RAND_pseudo_bytes(bp, padding);
29    r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,buffer,
30                          3 + payload + padding);
31    // [...]
32    if (r < 0) return r;
33   }
34   // [...]
35   return 0;
36 }
```

Fig. 1: The "Heartbleed" vulnerability in OpenSSL.

A prominent example of a taint-style vulnerability is the "Heartbleed" bug in OpenSSL discovered in 2014 [1]. Figure 1 shows the problematic code: The integer `payload` is defined by the macro `n2s` that reads a sixteen bit integer from a network stream (line 11). This integer then reaches the third argument of a call to `memcpy` without undergoing any sort of validation (line 25). In particular, it is not assured that `payload` is smaller or equal to the size of the source buffer `pl`, and hence, uninitialized heap memory may be copied to the buffer `bp`, that is then send out to the network via a call to `dtls_write_bytes` on line 29.

This example highlights the importance of identifying taint-style vulnerabilities, as well as some the difficulties involved. First, `n2s` is a macro used exclusively in the code base of OpenSSL, and thus can only be modeled in a search pattern if project-specific API functions are considered. Second, line 12 shows the check introduced to patch the vulnerability. The sanitization of `payload` is not trivial and difficult to analyze without profound knowledge of the code base.

To describe taint-style vulnerabilities in enough detail and to search for their incarnations in software, it is necessary to inspect how information propagates from one statement to

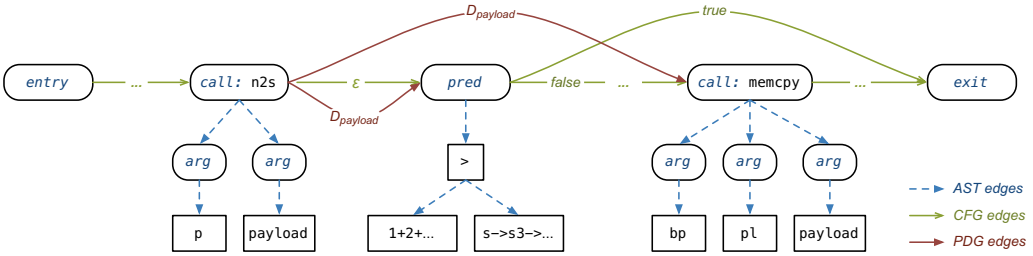


Fig. 2: An excerpt of the code property graph for the "Heartbleed" bug. Data-flow edges, control-flow edges and syntax edges are indicated by red, green, and blue color respectively.

another as well as how this flow is controlled by conditions. As a consequence, we base our method on code property graphs, an intermediate representation of source code that combines this information in a single structure and can be mined for vulnerabilities using graph traversals.

B. Code Property Graphs

There exists a large variety of representations for program code from the areas of software engineering and compiler design. For example, the structure of a program can be described as a syntax tree, while the order in which the program's statements are executed is captured in a control-flow graph. Several of these standard representations can be expressed as graphs and thus Yamaguchi et al. [63] propose to mine for vulnerabilities using graph databases. The main idea of their approach is to construct a so-called *code property graph*, a joint representation of a program's structure, control flow and data flow in a property graph—the native storage format of many graph databases. This joint representation enables programming patterns to be encoded as queries for graph databases, making it possible to mine large amounts of code for instances of dangerous programming patterns, and thus narrow in on vulnerabilities.

Formally, a property graph is an edge-labeled, attributed multigraph [46]. In practice, this means that key-value pairs can be attached to nodes and edges to store data. In addition, edges can be labeled to express different types of relationships in the graph. For example, for code analysis, nodes can be created for different language elements, such as calls, predicates and arguments. These can then be connected by labeled edges to represent execution order or data flow.

The code property graph makes use of this versatile data structure to combine three existing, well-understood program representations: the *abstract syntax tree*, which represents how program constructs are nested, the *control-flow graph*, which exposes statement execution order, and finally, the *program dependence graph*, which makes data-flow and control-dependencies explicit [see 3, 14]. Combining these representations in a property graph is possible as all of these representations contain designated nodes for each statement, allowing them to be merged easily at these nodes.

As an example, Figure 2 shows an excerpt of the code property graph for the function in Figure 1. The graph contains a node for each program statement, including the `entry` and `exit` statements, the call to `n2s` (line 11), the call to `memcpy` (line 25), and the if statement (line 12). Each of these nodes span a syntax tree indicated by blue edges, making, for example, the decomposition of the call to `n2s` into its language elements apparent. In addition, data-flow edges obtained from the program dependence graph are introduced from the first to the second and third statement to indicate that the value of `payload` produced at the first statement reaches the second and third unmodified and is used there. Finally, control-flow edges indicating the flow of control in the function are shown in green. For example, an unconditional control-flow edge connects the first statement to the second, making clear that the second statement is executed right after the first.

Once constructed, insecure programming patterns, and in particular, instances of taint-style vulnerabilities, can be described as *traversals* in the code property graph. Starting from a set of seed nodes, a traversal passes over the graph moving along the edges according to properties of the nodes. The output of a traversal is the set of nodes where this movement terminates. For example, a traversal may start at all calls to `memcpy` and move backwards along data-flow edges to the macro `n2s`, thereby extracting candidates similar to the "Heartbleed" vulnerability (Figure 1). Additionally, the traversal can make use of control-flow edges to only select those paths between `memcpy` and `n2s` where no validation of the propagated variable `payload` is performed.

Formally, graph traversals are functions that map one set of nodes to another set of nodes. As such, traversals can be chained freely using function composition to yield new traversals, making it possible to express complex queries in terms of re-usable, elementary traversals. We make extensive use of this capability to construct search queries for taint-style vulnerabilities (see Section IV-D). A detailed introduction to code property graphs and traversals for vulnerability discovery is provided by Yamaguchi et al. [63].

III. Extending Code Property Graphs for Interprocedural Analysis

The code property graph offers a wealth of information for pattern-based vulnerability discovery, however, it has not been constructed with interprocedural analysis in mind. Unfortunately, information exploitable to infer search patterns is scarce

and often spread across several functions, making analysis beyond function boundaries desirable. We therefore seek to extend the code property graph to obtain a representation similar to the well known System Dependence Graph [23] but in a format suitable for mining using graph databases. We can achieve this by extending the code property graph as follows.

We begin by making the data flow between call sites and their callees explicit by introducing edges from arguments to parameters of the respective callees, and from return statements back to call sites. In effect, we already obtain a graph that expresses call relations between functions, however, the data flow information it encodes is needlessly inexact. Most importantly, modifications made by functions to their arguments are not taken into account, nor the effects these have as data flows back along call chains.

In the following, we describe an approach to improve this preliminary graph by detecting argument modifications using post-dominator trees (Section III-A), both to handle calls to functions where source code is available, and for library functions where only callers can be observed (Section III-B). We proceed to propagate this information through the graph to obtain the final interprocedural version of the code property graph used for inference of search patterns (Section III-C).

A. Adding Post-Dominator Trees

For our heuristic approach to the detection of argument definitions, the ability to determine whether a statement is always executed before another is crucial. Unfortunately, the existing classic program representations merged into the property graph do not allow this to be determined easily; the control-flow graph only indicates whether a statement *may* be executed after another, and the control-dependencies of the program dependence graph are limited to exposing *predicates* that must be evaluated before executing statements.

Dominator and post-dominator trees [see 3, 8], two classical program representation derivable from the control-flow graph, are ideally suited to address this problem. As is true for control-flow graphs and program dependence graphs, these trees contain a node for each statement. These nodes are connected by edges to indicate *dominance*, a notion closely related to analysis of mandatory statement execution-order.

A node d dominates another node n in a control-flow graph, if every path to n has to first pass through d. By linking each node to its immediate dominator, we obtain a dominator tree. Similarly, a node p post-dominates another node n, if every path from n has to pass through p. By again linking together the immediate post-dominators of each node, we obtain a *post-dominator tree*.

As an Example, Figure 4 shows a post-dominator tree for the function `bar` of the running example from Figure 3. Like all post-dominator trees, the tree is rooted at the exit node as all paths in the CFG eventually lead through the exit node. However, as edges only exist from nodes to their immediate post-dominator, only the predicate `y < 10` and the call `foo(x,y,z)` are connected to the exit node. In contrast,

```
int bar(int x, int y) {
     int z;
     boo(&z);
     if (y < 10)
       foo(x,y,&z);
 }

 int boo(int *z) {
     *z = get();
 }

int moo() {
     int a = get();
     int b = 1;
     bar(a, b);
 }

 int woo() {
     int a = 1;
     int b = get();
     bar(a, b);
 }
```

Fig. 3: Running example of a call to the sink `foo`

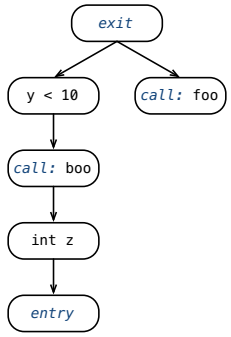


Fig. 4: Post-dominator tree for the function `bar`

the call to `boo` is immediately post-dominated by the predicate, while it immediately post-dominates the statement `int z`.

Both data structures come handy, if we quickly need to determine whether a statement in the code base is always preceded or followed by another statement. Moreover, since a designated node exists for each statement, merging these structures with the existing code property graph can be easily achieved by adding appropriately labeled dominator edges between statement nodes.

B. Detecting Argument Modification

Once post-dominator trees are available, we can employ them to detect function calls that result in modifications of their arguments —a process denoted as *defining arguments* in compiler design [3]. While for common library functions such as `read` or `recv` from the POSIX standard, this problem can be addressed by providing annotations, internal APIs such as the `n2s` macro as present in the "Heartbleed" vulnerability are not recognized as input sources.

In general, we therefore need to assume that it is unknown whether a call to a library function results in modification of its arguments, and hence, serves as a data source. In effect, for all direct and indirect callers of library functions, argument

definition may not be detected correctly. As an example, consider the POSIX library functions `read` and `write` that both take a pointer to a memory buffer as their second arguments. From the function signatures alone, it is impossible to determine that `read` modifies the contents of the buffer while `write` does not. This, however, is a vital difference that directly affects data-flow edges in the code property graph.

To address this problem, we proceed as follows. For each encountered function that comes without source code, we determine whether it possibly defines its arguments by calculating a simple statistic based on the following two checks.

1) We check whether a local variable declaration reaches the argument via a data flow without undergoing an easily recognizable initialization, such as an assignment or a call to a constructor.
2) We check that the path from the function to the local variable declaration in the post-dominator tree does not contain another statement that is also directly connected to the variable declaration by data flow.

Calculating the fraction of call sites that fulfill both conditions, we assume that an argument is defined by calls to the function if the fraction is above a defined threshold. For our experiments we fix this threshold to 10% to not miss any attacker-controlled sources. With the help of this simple heuristic, we recalculate the data-flow edges in the code property graph for all callers of functions without source code.

Figure 3 illustrates our heuristic: the local variable `z` is declared on line 2 without obvious initialization. It is then passed to both `boo` and `foo` as an argument on line 3 and 5, respectively. While for function `boo`, it is reasonable to assume that it initializes `z`, this is not true for function `foo` as it is called after `boo` that may have already initialized `z`.

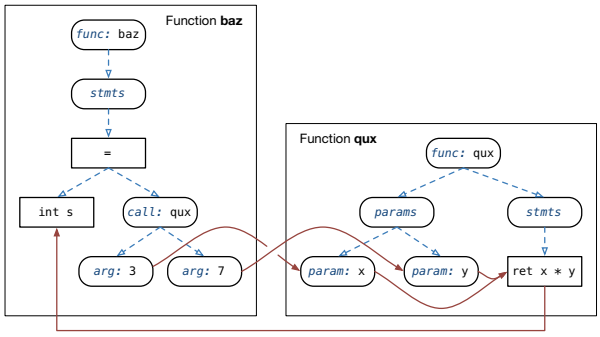
C. Propagation of Data-Flow Information

In addition to the problem of detecting argument definitions by library functions, argument definitions may occur indirectly, i.e., any of the functions called by a function may be responsible for argument definition. Consequently, identifying the sources for a data flow in a function without descending into all its callees is not effective. As an example, consider the code snippet shown in Figure 3. The argument `z` of the function `foo` is first defined in line 2 but then re-defined in line 9 inside the called function `boo`. As a result, there is a data flow from the source `get` to the function `foo`.

To take into account indirect argument definitions, we can propagate data-flow information along call-chains. To this end, we determine for each function (with available source code) whether argument definition takes place by analyzing its body and checking whether any of its parameters are (a) defined inside the function and (b) this definition reaches the exit statement via control flow. However, this only works if the data-flow edges of a function already take into account argument definitions performed by any of the functions that it calls. We therefore analyze all callees of a function prior to analyzing the function itself and apply the heuristic presented in the previous section for library functions.

Algorithm 1 Data flow recalculation

```
1:  procedure FixDataFlowEdges(V)
2:      for v ∈ V do
3:          f_v ← false                          ▷ Mark nodes as not fixed
4:      for v ∈ V do
5:          FixNode(v)
6:  procedure FixNode(v)
7:      if f_v = true then
8:          return false
9:      f_v ← true, u ← false
10:     for c ∈ Callees(v) do                    ▷ Fix all callees
11:         u ← u ∨ FixNode(c)
12:     if u = true then                         ▷ v needs to be updated
13:         UpdateDataFlow(v)
14:         return true
15:     return false
```



(a) Interprocedural code property graph.

```
1  void baz(void) {
2      int s = qux(3, 7);
3  }

4  int qux(int x , int y) {
5      return x * y;
6  }
```

(b) Code snippet of caller and callee.

Fig. 5: Interprocedural code property graph for the functions `baz` and `qux`. Syntax edges are shown as dotted lines and data-flow edges as solid lines.

Algorithm 1 implements this idea by recursively updating the nodes of the code property graph. In particular, the data-flow edges of a node v are fixed using the procedure FixNode, where the attribute f_v ensures that no node is visited twice. The algorithm descends into the graph using a pre-order traversal, that is, all callees are updated (line 11) before the current function is processed (line 13). Upon completion of Algorithm 1, observable argument definitions and the resulting indirect data flows are accounted for in the graph.

As an example of a resulting interprocedural code property graph, consider the code snippet and the graph given in Figure 5. The graph is constructed from the nodes of the abstract syntax tree, where its edges either reflect syntax (dashed lines) or data flow (solid lines). Control-flow edges are not shown in this example.

Conceptually, this interprocedural representations of code is directly derived from the classical System Dependence Graph (SDG) introduced by Horwitz et al. [23], however, tuned to

be processed using graph database queries, and augmented with syntax and dominance information. This structure is well suited to model and search for vulnerabilities as we illustrate in the following sections.

IV. Inference of Search Patterns for Vulnerabilities

Equipped with a code property graph extended for interprocedural analysis, we are now ready to tackle the problem of extracting search patterns for taint-style vulnerabilities. Starting from a security-sensitive sink, such as the memory function `memcpy`, our goal is to generate search patterns in the form of graph traversals that enable uncovering vulnerabilities in the data flow to the sink. To be useful, these queries need to be general enough to encode patterns of code instead of specific invocations. Moreover, they need to capture data flow precisely across functions, such that the definition of individual arguments can be correctly tracked. Finally, the generated queries should be easy to understand and amendable by a practitioner, allowing additional domain knowledge to be incorporated.

In order to generate search patterns with these qualities, we implement the following four-step procedure that combines techniques from static code analysis, machine learning and signature generation (Figure 6).

1) *Generation of definition graphs.* For each call of the selected sink, we generate *definition graphs* by analyzing the code property graph. Similar to interprocedural program slices [23], these graphs compactly encode both argument definitions and sanitizations, albeit in a two-level structure created specifically to easily enumerate feasible invocations (Section IV-A).
2) *Decompression and clustering.* We then decompress the definition graphs into individual invocations and cluster these for all call sites to obtain patterns of common argument definitions (Section IV-B).
3) *Creation of sanitization overlays.* Next, we extend the generated patterns by adding potential sanitization from the data flow, that is, all conditions in the flow restricting the argument values (Section IV-C).
4) *Generation of graph traversals.* Finally, we express the inferred search patterns in the form of graph traversals suitable for efficient processing using the analysis platform Joern (Section IV-D).

In the following sections, we describe each of these steps in more detail and illustrate them with examples.

A. Generation of Definition Graphs

While source code contains valuable information about how functions are invoked, and in particular, how their arguments are defined and sanitized, this information is often spread across several different functions and buried in unrelated code. To effectively exploit this information, a source-sink representation is required that encodes *only* the definitions and sanitizations of arguments, while discarding all other statements. To address this problem, we generate a graph representation for each source-sink system that can be easily calculated from our code property graphs. This representation, referred to as a *definition graph* throughout the paper, encodes all observed combinations of argument definitions and their corresponding sanitization. As such, definition graphs are created from a carefully chosen subset of the nodes of the corresponding interprocedural program slices [see 23, 60], containing only nodes relevant for determining search patterns for taint-style vulnerabilities. Definition graphs allow to easily enumerate feasible argument initializations, as is possible for complete interprocedural program slices by solving a corresponding context-free-language reachability problem [see 44].

We construct these graphs by first modeling individual functions locally, and then combining the respective graphs to model function interaction.

1) Local Function Modeling: Within the boundaries of a function, determining the statements affecting a call and the variables involved can be achieved easily using program slicing techniques. This allows us to create a hierarchical representation that captures all definition statements involving variables used in the call as well as all conditions that control the execution of the call site. To illustrate the construction of such a representation, we consider the call to the function `foo` (line 5) in Figure 3 as a selected sink. Starting from this call site, we construct the representation by passing over the code property graph using the following rules:

- For the selected sink, we first follow the outgoing syntax edges to its arguments. In the example, we expand `foo` to reach the arguments `x`, `y` and `z`.
- For these arguments and all statements defining them, we then follow connected data-flow and control-dependence edges to uncover defining statements as well as conditions that control the call site `foo`. For example, the definition `int z` and the condition `y < 10` are discovered in this way.
- Finally, we use interprocedural edges to move from all calls that define any of our variables to the respective function bodies, where we identify further defining statements that affect the arguments of the selected sink. In the example, the call to `boo` is discovered by this rule, leading us to the definition statement `*z = get()`.

For each parameter reached in this way, we consider all respective call sites as sinks, and recursively apply these rules to obtain a tree for each function connected with the initial sink via data flow. These trees can be easily constructed from the code property graph using a depth first traversal that employs an expansion function implementing the three rules given.

2) Definitions graphs: With the tree representations of functions at hand, we can already analyze argument definitions and their sanitization within a function, however, callers remain unexplored. In the example code, this means that we uncover the local variable definition of `z`, while the parameters `x` and `y` cannot be traced past the function boundary. Unfortunately, the simple and intuitive solution of following parameter-to-argument edges during the construction of local

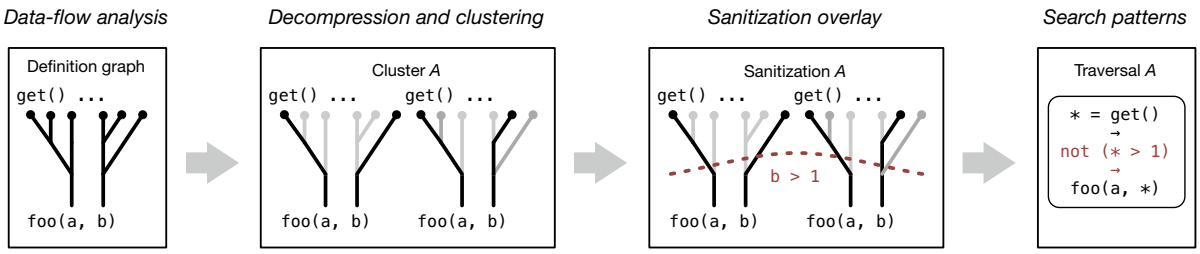


Fig. 6: Overview of our method for inference of search patterns for vulnerabilities. Starting from a selected sink (`foo`), the method automatically constructs patterns that capture sources (`get()`) and sanitization (`b > 1`) in the data flow.

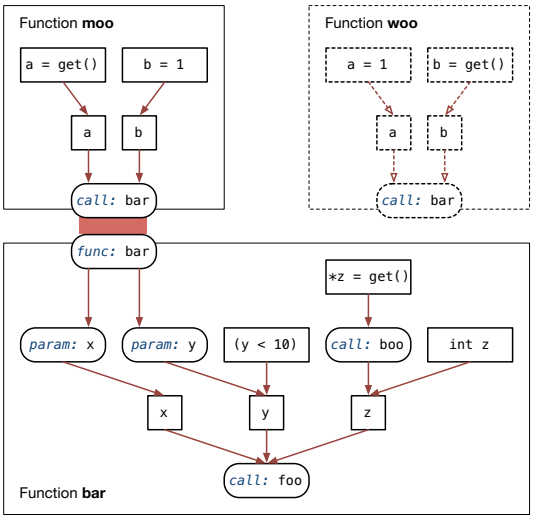


Fig. 7: Definition graph for the function `foo` of the running example from Figure 3 with arguments defined by `moo`. A second instantiation of the graph is shown with dashed lines for `woo`.

trees produces infeasible combinations of definitions as discussed in detail by Reps [44]. For instance, in our example, this simple solution would generate the combination {`int a = get(), int b = get()`}. However, this combination is invalid as the first definition only occurs when `moo` calls `bar` while the second occurs when `woo` calls `bar`. Hence, these definitions never occur in combination.

This is a classical problem of interprocedural program analysis, which can, for instance, be solved by formulating a corresponding context-free-language reachability problem [44]. Another solution is to simply ensure that parameter nodes of a function are always expanded together when traversing the graph. For example, when expanding the parameter node for `x`, the node for `y` needs to be expanded as well. Moreover, it needs to be ensured that both nodes are expanded with arguments from the same call site, in our example either `woo` or `moo`.

As solution we simply tie parameters together by modeling the interplay of entire functions as opposed to parameters. The *definition graph* implements this idea. In contrast to the trees modeling functions locally, nodes of the definition graph are not simply a subset of the nodes of the interprocedural code property graph, but represent entire trees. Definition graphs are therefore two-level structures that combine trees used to model functions locally to express their calling relations. As an example, Figure 7 shows the definition graph for the call to `foo` in the sample code. Formally, we can define these definition graphs as follows.

Definition 1. A *definition graph* $G = (V, E)$ for a call site c is a graph where V consists of the trees that model functions locally for c and those trees of all of its direct and indirect callers. For each $a, b \in V$, an edge from a to b exists in E if the function represented by a calls that represented by b.

B. Decompression and Clustering

We now have a source-sink representation that makes the definition of arguments and their sanitization explicit. We thus seek to determine patterns in the definition graphs that reflect common combinations of argument definitions. Given an arbitrary set of definition graphs, for example all definition graphs for all call sites of the function `memcpy`, we employ machine learning techniques to generate clusters of similar definition combinations along with their sanitizers, designed to be easily translated into graph database traversals (see Section IV-D). We construct these clusters in the following three steps.

1) Decompression of definition graphs: While a definition graph represents only a single sink, it possibly encodes multiple combinations of argument definitions. For example, the definition graph in Figure 7 contains the combination {`int z, a = get(), b = 1`} as well as the combination {`int z, a = 1, b = get()`} in a compressed form. Fortunately, enumerating all combinations stored in a definition graph can be achieved using a simple recursive procedure as shown in Algorithm 2, where $[v_0]$ denotes a list containing only the node v_0 and the operator $+$ denotes list concatenation.

The nodes of the definition graph are trees, where each combination of argument definitions corresponds to a subset of these nodes that represent a call chain. Starting from the root node $r(V)$, the algorithm thus simply combines the current tree with all possible call chains, that is, all lists of trees

Algorithm 2 Decompression of definition graph

```
1: procedure DECOMPRESS(G)
2:     return RDECOMPRESS(G, r(V))
3: procedure RDECOMPRESS(G := (V, E), v_0)
4:     R = ∅
5:     D ← PARENTTREES(v_0)
6:     if D = ∅ then
7:         return {[v_0]}
8:     for d ∈ D do
9:         for L ∈ RDECOMPRESS(G, d) do
10:            R ← R ∪ ([v_0] + L)
11:    return R
```

encountered in the code base that lead to this tree. As a result of this step, we obtain the set of all observed combinations of argument definitions denoted by S.

2) Clustering callees and types: Callees and types with a similar name often implement similar functionality. For example, the functions `malloc` and `realloc` are both concerned with allocation, while `strcpy` and `strcat` deal with copying strings into a buffer. We want to be able to detect similar combinations of definitions even if none of the arguments are defined using exactly the same callee or type. To achieve this, we determine clusters of similar callees and types prior to constructing search patterns for vulnerabilities.

In particular, we cluster the callees and types for each argument independently. As we are interested in compact representations, we apply *complete-linkage clustering*, a technique that is known for creating compact groups of objects and easy to calibrate [see 4]. Linkage clustering requires a distance metric to be defined over the considered objects, and we employ the Jaro distance for this task, as it has been specifically designed for comparing short strings [25]. The Jaro distance quantifies the similarity of two strings as a value between 0 and 1, where a value of 1 indicates an exact match and a value of 0 indicates absolute dissimilarity. The analyst can control how similar strings inside a cluster need to be by specifying a minimum similarity in terms of the Jaro distance. We found that the clustering is relatively stable for clustering parameters between 0.9 and 0.7 and fixed the parameter to 0.8 for all of our experiments. As a result of this step, we obtain a set C of callee and type clusters for each argument.

3) Clustering of combinations of definitions: Clustering the decompressed combinations of argument definitions is slightly more involved than clustering callees and types as these combinations are complex objects rather than simple strings. Our goal is to compare definition combinations in terms of the definitions they attach to arguments, albeit in a way robust to slight differences in the names of callees and types. We achieve this by using a generalized bag-of-words model and mapping the combinations to a vector space spanned by the clusters calculated in the previous step [see 45].

In the following, let us assume a sink with a single argument and let S denote the set of combinations while C is the set of callee and type clusters. Then, for each combination $s \in S$, we can determine the clusters $C_s \subseteq C$ that its definitions are contained in. We then represent each $s \in S$ by a vector in a space where each dimension is associated with one of the clusters of C. We can achieve this by defining a map $\phi : S \mapsto \{0,1\}^n$ where the c'th coordinate ϕ_c is given by

$$\phi_c(s) = \begin{cases} 1 & \text{if } c \in C_s \\ 0 & \text{otherwise} \end{cases}$$

and n is the total number of callee and type clusters $|C|$.

In the case of sinks with multiple arguments, we perform this operation for each argument independently and simply concatenate the resulting vectors. As an example, let us consider a definition combination s where the first argument is initialized via a call to `malloc` while the second is defined to be of type `size_t`. Then the corresponding vector has the following form.

$$\phi(s) \mapsto \begin{pmatrix} \dots & \dots & \\ 0 & \{\texttt{char[52]}, \texttt{uchar[32]}, \dots\} & \} \text{ Arg. 1} \\ 1 & \{\texttt{malloc}, \texttt{xmalloc}, \dots\} & \\ \dots & \dots & \\ 1 & \{\texttt{size\_t}, \texttt{ssize\_t}, \dots\} & \} \text{ Arg. 2} \\ 0 & \{\texttt{int}, \texttt{uint32\_t}, \dots\} & \\ \dots & & \dots \end{pmatrix}$$

Using this vectorial representation, we can now employ linkage clustering again to obtain clusters of similar combinations for argument definitions. As a distance function for the clustering, we choose the city-block distance, since it provides an intuitive way for measuring the presence or absence of clusters in the vectors. We use a fixed value of 3 for the clustering parameter throughout our experiments, meaning that invocations inside a cluster may differ in up to three entries. As a result of this step, we obtain clusters of similar combinations, i.e., groups of similar invocations that constitute patterns present in the code base. The size of these clusters can be used to rank these patterns in order to prioritize inspection of dominant patterns: larger clusters represent strong patterns supported by many individual invocations, while small clusters represent less articulate patterns, supported only by few invocations.

C. Creation of Sanitization Rule Overlays

The clusters generated in the previous step can already be used to generate search patterns indicating the combinations of arguments predominant for a sink. They do not, however, encode sanitization patterns. To achieve this, we proceed to create overlays for argument definition models that express typical sanitization of each argument. To this end, we exploit the information spread across all *conditions* contained in any of the definition graphs. As is the case for callee and type matching, we seek to be robust against slight variations in the way conditions are formulated. To this end, we make use of the fact that each condition is represented as a syntax tree in the code property graph. Similar to the way callees and types are grouped, we map these trees to vectors and cluster them using linkage clustering. In particular, we employ an explicit tree embedding based on the neighborhood hash kernel [16, 22].

Let $\mathcal{T}$ denote the set of conditions, each represented by a syntax tree. Then we map each condition $t \in \mathcal{T}$ to a vector as follows: First, inner nodes in the syntax tree are labeled by the hash value of their `type` attributes, e.g., multiplication, unary expression or assignment, while leaf nodes are labeled by the hash value of their `code` attribute, typically an identifier, operator or literal. Second, the symbol that is being propagated is renamed to `var`. In that way, the condition does not depend on the name of the identifier. Numbers, relational, and equality operators are also normalized using regular expressions. Finally, the neighborhood of each node v is taken into account by computing its *neighborhood hash* as a function of the labels of its child nodes as

$$h(v) = r(l(v)) \oplus \left(\bigoplus_{z \in C_v} l(z) \right)$$

where $l(v)$ is the label of the node v, $r(\cdot)$ denotes a single-bit rotation, $\oplus$ indicates a bit-wise XOR operation on the labels of the nodes, and finally C_v are the child nodes of v. In effect, we obtain a set of hash values for each syntax tree that we can use to represent it. We define the mapping $\Phi : \mathcal{T} \to \{0,1\}^n$ from conditions to n-dimensional vectors where n is the number of different hash values and for a condition $c \in \mathcal{T}$

$$\Phi_j(c) = \begin{cases} 1 & \text{if } c \text{ contains a node } v \text{ with } h(v) = j \\ 0 & \text{otherwise} \end{cases}.$$

Upon performing this mapping for each condition in the defnition graph, we employ linkage clustering yet again using the city-block distance with a fixed parameter of 2, yielding clusters of conditions. We store the cluster identifiers of all conditions used in each of the combination clusters calculated in the previous section and finally attach them to the search patterns.

D. Generation of Graph Traversals

The generated clusters enhanced with sanitization overlays fully express search patterns that can now finally be mapped to graph traversals to mine for vulnerabilities. To achieve this, we construct a generic template for the pattern-based analysis platform Joern that can capture missing sanitization of arguments and can be easily instantiated to express different search patterns for taint-style defects.

1) Traversal template: Figure 8 shows the template traversal in the query language Gremlin. To instantiate the template, we need to define the name of the sensitive sink, descriptions for data sources of each argument as well as descriptions for their sanitization. These are referred to as `argiSource` and `argiSanitizer` where i denotes the argument number. The traversal proceeds by determining all call sites of `sink`. It then processes each sink separately using the traversals `taintedArgs` and `unchecked` in order.

The traversal `taintedArgs` is used to find if a sink conforms to the source descriptions (`argiSource`). It achieves this by first generating the corresponding definition graph as described in Section IV-A. Without decompressing the graph, it then determines whether the call site can possibly fulfill the argument description by checking whether for each description, at least one matching statement exists in the definition graph.

```
getCallsTo(sink)

.taintedArgs(
        [arg1Source,..., argnSource]
)

.unchecked(
        [arg1Sanitizer, ... argnSanitizer]
)
```

Fig. 8: Template for taint-style vulnerability as a graph traversal in the query language Gremlin.

This step drastically reduces the number of call sites that need to be analyzed further, however, we cannot yet tell with certainty whether the call site matches the argument descriptions. To achieve this, the particular definition graph is decompressed according to Algorithm 2. With definition combinations at hand, it now becomes trivial to check whether argument descriptions are matched. Finally, we thus return all definition combinations that match the descriptions and pass them on to the traversal `unchecked` for analysis of sanitization.

The traversal `unchecked` determines all call sites where at least one of the arguments is not sanitized according to the sanitizer descriptions. The function proceeds by checking each of the conditions in the definition graph against the respective sanitizers descriptions.

2) Template instantiation: We instantiate queries from clusters by translating definitions and conditions into argument descriptions and sanitizer descriptions respectively. Recalling that for each argument, a set of clusters for data sources as well as conditions is available, these merely have to be summarized in a form suitable to be understood easily by security analysts. To this end, we generate regular expressions from these clusters by determining longest common sub sequences as commonly performed in signature generation [see 40]. The resulting graph traversals can then be used as is to mine code for bugs as well as allow for refinement by analysts.

V. Evaluation

We proceed to evaluate our method on the source code of five popular open-source applications in two steps. First, we perform a controlled experiment where we evaluate our method's ability to generate traversals for known vulnerabilities and measure how these reduce the number of call sites to inspect. Second, we evaluate our method's ability to assist in a real-world code audit of the popular media player VLC where we uncover several previously unknown vulnerabilities. We implement our method as a plugin for the code analysis platform *Joern* version 0.3.1, using the library *fastcluster* [37] for clustering. To allow other researchers to reproduce our results, we make our plugin available as open source[2].

[2]https://github.com/fabsx00/querygen

Project	Version	Component	Lines of code	Vulnerability	Sensitive sink	# Call Sites
Linux	3.11.4	Driver Code	6,723,955	CVE-2013-4513	`copy_from_user`	1715
OpenSSL	1.1.0f	Entire Library	378,691	CVE-2014-0160	`memcpy`	738
Pidgin	2.10.7	Entire Program	363,746	CVE-2013-6482	`atoi`	255
VLC	2.0.1	Entire Program	555,773	CVE-2012-3377	`memcpy`	879
Poppler (Xpdf)	0.24.1	Entire Library	227,837	CVE-2013-4473	`sprintf`	22

TABLE I: Data set of five open-source projects with known taint-style vulnerabilities. The table additionally lists the sensitive sinks of each vulnerability and the number of traversals inferred by our method.

	Correct Source	Correct Sanitization	# Traversals	Generation Time	Execution Time	Reduction[%]
CVE-2013-4513	✓	✓	37	142.10 s	10.25 s	96.50
CVE-2014-0160	✓	✓	38	110.42 s	8.24 s	99.19
CVE-2013-6482	✓	✓	3	20.76 s	3.80 s	92.16
CVE-2012-3377	✓	✓	60	229.66 s	20.42 s	91.13
CVE-2013-4473	✓		1	12.32 s	2.55 s	95.46
Average						94.90

TABLE II: Reduction of code to audit for discovering the five taint-style vulnerabilities. For the last vulnerability no correct sanitizer is infered due to the low number of call sites.

A. Controlled Experiment

To evaluate our method's ability to generate queries for real vulnerabilities in program code in a controlled setting, we analyze the security history of five popular open-source projects: the Linux kernel, the cryptographic library OpenSSL, the instant messenger Pidgin, the media player VLC and finally, the rendering library Poppler as used by the document viewers Evince and Xpdf. For each of these projects, we determine a recent taint-style vulnerability and the associated sensitive sink. Table I provides an overview of this data set, showing the project and its version, the vulnerable component, and the lines of code it contains. Moreover, the vulnerability, denoted by its CVE-identifier, the associated sensitive sink, and the number of call sites of the sink are shown. We now briefly describe each of these taint-style vulnerabilities in detail.

- *CVE-2013-4513 (Linux).* An attacker-controlled variable named `count` of type `size_t` is passed as a third argument to the sink `copy_from_user` without being sanitized, thereby triggering a buffer overflow.
- *CVE-2014-0160 (OpenSSL "Heartbleed").* The variable `payload` of type `unsigned int` as defined by the source `n2s` is passed as a third argument to `memcpy` without being checked, causing a buffer overread.
- *CVE-2013-6482 (Pidgin).* The string `unread` is read from the attacker-controlled source `xmlnode_get_data` and passed to the sink `atoi` without undergoing sanitization, thereby possibly causing a NULL pointer to be dereferenced.
- *CVE-2012-3377 (VLC).* The length of the data buffer `p_stream->p_headers` is dependent on an attacker-controlled allocation via the function `realloc` and reaches a call to `memcpy` without verifying the available buffer size, leading to a buffer overflow.
- *CVE-2013-4473 (Poppler).* The attacker-controlled string `destFileName` is copied into the local stack buffer `pathName` of type `char [1024]` using the function `sprintf` without checking its length, leading to a stack-based buffer overflow.

We proceed to generate traversals for all of these sinks. Table II summarizes our results, showing the number of traversals generated for each vulnerability, and whether our method was able to generate a traversal that expresses both the correct source and sanitizer. It also shows the time required to generate traversals from the code, and the execution time of the traversal in seconds. Finally, the percentage of call sites that do not have to be inspected when using the generated traversal as a robust signature for the vulnerability is shown (reduction percentage).

Our method generates correct descriptions for the respective argument sources in all cases, and correct sanitizers in all but one case. In the case of CVE-2013-4473, no sanitizer description is returned. In this case, only 22 call sites are available, making the inference of a sanitizer description difficult using statistical methods. Regardless of this, the number of call sites to inspect to locate the vulnerabilities is drastically reduced by our queries, allowing 94.9% of the call sites to be skipped on average.

Finally, Table III shows the inferred regular expressions for sources and sinks. In these regular expressions the names of attacker-controlled sources from the vulnerability descriptions are clearly visible. Moreover, apart from those sanitization patterns from the bug descriptions, additional sanitizers are recognized in some cases. For example, the method determines that the first argument to `memcpy` stemming from the source `n2s` is commonly compared to `NULL` to ensure that it is not a `NULL` pointer. For arguments where multiple sanitizers are enforced, only one is shown.

	CVE-2013-4513	CVE-2014-0160	CVE-2013-6482	CVE-2012-3377	CVE-2013-4473
Sink	`copy_from_user`	`memcpy`	`atoi`	`memcpy`	`sprintf`
Argument 1	`.*`	`.*`	`.*xmlnode_get_.*`	`.*alloc.*`	`.*char \[ .* \].*`
Argument 2	`.*const .*cha.*r *.*`	`.*`	`.*`	`.*`	`.*`
Argument 3	`.*size_t.*`	`.*n2s.*`	`.*`	`.*`	`.*`
Sanitizer 1	–	`.*sym ( ==|!=) NULL.*`	`.*sym.*`	`.*sym.*`	–
Sanitizer 2	–	–	–	–	–
Sanitizer 3	`.*sym .*(\d+).*`	`.*sym.*\+(\d+).*`	–	–	–

TABLE III: Regular expressions contained in the search patterns for the five taint-style vulnerabilities, where *sym* is replaced by the tracked symbol at runtime. For the last vulnerability, no sanitizers are inferred.

B. *Case Study: The Heartbleed Vulnerability*

In this case study, we show how our method successfully generates a search pattern for the "Heartbleed" vulnerability presented in Section II as an example of a taint-style vulnerability. We use our method to generate patterns for the security-sensitive sink `memcpy` in OpenSSL version 1.1.0f, the last version of the library to be vulnerable to this particular bug. Among functions such as `strcpy`, `strcat` and `sprintf`, `memcpy` is one of the functions most commonly associated with buffer overflow vulnerabilities [see 5, 11].

We begin by employing the heuristic presented in Section III-B to discover library functions that define their arguments. Figure 9a shows the discovered library function names and argument numbers. We manually verify each of these to find that all but one are inferred correctly. For the falsely identified third argument to `memset`, we found that it is often of the form `sizeof(buffer)` where buffer is a variable reaching `memset` without prior definition. Slightly adapting our heuristic to account for the semantics of `sizeof` (by suppressing its arguments) fixes this problem as well, leaving us only with correctly inferred data sources.

Function	Defining
fgets	1. argument
sprintf	1. argument
memset	1. argument
write	3. argument
memcpy	1. argument
memset*	3. argument
n2s	2. argument
n2l	2. argument
c2l	2. argument

(a) Library functions

Regular expression
.*n2s.*
.*memset.*
.*strlen.*
.*int .*len.*
.*int arg.*
.*size_t.*
.*unsigned.*
.*int.*
.*long.*

(b) Inferred data sources

Fig. 9: (a) Tainting library functions identified by our heuristic; (b) regular expressions inferred for data sources of the third argument to `memcpy`.

Inferring queries from the code then leads to the generation of 38 queries, 14 of which specify a source for the third argument of `memcpy`. These are particularly interesting as the third argument specifies the amount of data to copy and hence cases where it is attacker-controlled are prime candidates for buffer overflows. Figure 9b contains the 9 sources of the third argument, showing the attacker-controlled source `n2s` in particular. As `n2s` is the only source that is under attacker control with certainty, only the single traversal shown in Figure 10 needs to be executed.

```
arg3Source = sourceMatches('.*n2s.*');

arg2Sanitizer = { it, symbol ->
     conditionMatches(".*%s (==|!=) NULL.*", symbol)
};

arg3Sanitizer = { it, symbol ->
     conditionMatches(".*%s.*\+(\d+).*", symbol)
};

getCallsTo("memcpy")
.taintedArgs([ANY, ANY, arg3Source])
.unchecked([ANY_OR_NONE, arg2Sanitizer, arg3Sanitizer])
```

Fig. 10: Generated traversal encoding the vulnerable programming pattern leading to the Heartbleed vulnerability.

The query encodes the flow of information from the attacker-controlled data source `n2s` to the third argument of `memcpy`. Moreover, it enforces two sanitization rules. First, it needs to be checked whether the second argument passed to `memcpy` is a NULL pointer and second the third argument needs to be checked in an expression containing an integer. Clearly, these rules can be easily modified by an analyst to increase precision; however, for the purpose of this case study, we employ the traversal as is. Even without refinement the traversal returns only 7 call sites of 738 (0.81%) shown in Table IV. Among these, two correspond exactly to the "Heartbleed" vulnerability.

C. *Case Study: Vulnerabilities in the VLC Media Player*

In this case study, we illustrate how our method plays the key role in the identification of five previously unknown

Filename	Function
ssl/d1_both.c	dtls1_process_heartbeat
ssl/s3_clnt.c	ssl3_get_key_exchange
ssl/s3_clnt.c	ssl3_get_new_session_ticket
ssl/s3_srvr.c	ssl3_get_client_key_exchange
ssl/t1_lib.c	ssl_parse_clienthello_tlsext
ssl/t1_lib.c	tls1_process_heartbeat
crypto/buffer/buf_str.c	BUF_memdup

TABLE IV: The seven hits returned by the generated query. Vulnerable functions are shaded.

Traversal	Filename	Function	Line	CVE Identifier
Traversal 1	modules/services_discovery/sap.c	ParseSDP	1187	CVE-2014-9630
Traversal 1	modules/stream_out/rtpfmt.c	rtp_packetize_xiph_config	544	CVE-2014-9630
Traversal 1	modules/access/ftp.c	ftp_SendCommand	122	CVE-2015-1203
Traversal 2	modules/codec/dirac.c	Encode	926	CVE-2014-9629
Traversal 2	modules/codec/schroedinger.c	Encode	1554	CVE-2014-9629

TABLE V: The call sites extracted by our traversals. All of these call sites are vulnerable.

```
arg1Src = sourceMatches('.*char \[ .*len \+ .* \].*')
arg3Src = { sourceMatches('.*size_t.*')(it) ||
            sourceMatches('.*str.*len.*')(it)}

getCallsTo("memcpy")
.taintedArgs([arg1Src, ANY_SOURCE, arg3Src])
```

Fig. 11: Traversal to identify dynamic allocation of stack memory for the first argument of memcpy.

vulnerabilities in VLC, a popular open-source media player. To this end, we chose two of the traversals generated for the sink `memcpy` that look particularly interesting as they directly encode dangerous programming practices. The first query, shown in Figure 11, describes a call to `memcpy` where the first argument is defined to be a local stack buffer of type `char`. Moreover, the size is dynamically calculated inside the definition. This alone already constitutes a problematic programming practice as it is impossible to verify whether the available stack memory allows this allocation to be performed. In particular, if the amount of memory to be allocated is controlled by an attacker, and memory is subsequently copied into the buffer using `memcpy`, attackers can possibly corrupt memory and leverage this to execute arbitrary code.

Running this query returns three call sites, all of which are problematic. In particular, Figure 13 shows the vulnerable function `rtp_packetize_xiph_config` where, on line 14, the variable `len` is calculated to be the length of an attacker-controlled string. It is then used to allocate the stack buffer `b64` on line 15, and finally, on line 16, `len` bytes are copied to the buffer. The presence of this vulnerability has been successfully confirmed by triggering an invalid memory access on a 64 bit Linux platform.

Figure 12 shows a second interesting query: in this case, the second argument of `memcpy` stems from a source matching the regular expression `.*Get.*`. This corresponds to a family of macros in the VLC media player that read directly from media files possibly controlled by attackers. Cases where the amount of data to be copied into a buffer are directly dependent on

```
arg20Source = sourceMatches('.*Get.*');
arg21Source = sourceMatches('.*uint.*_t.*');

getCallsTo("memcpy")
.taintedArgs([ANY_SOURCE,ANY_SOURCE,
              {arg20Source(it) && arg21Source(it)}])
```

Fig. 12: Traversal to identify third arguments of `memcpy` defined by `.*Get.*`.

```
1  int rtp_packetize_xiph_config( sout_stream_id_t *id,
2                                 const char *fmtp,
3                                 int64_t i_pts )
4  {
5    if (fmtp == NULL)
6      return VLC_EGENERIC;
7
8    /* extract base64 configuration from fmtp */
9    char *start = strstr(fmtp, "configuration=");
10   assert(start != NULL);
11   start += sizeof("configuration=") - 1;
12   char *end = strchr(start, ';');
13   assert(end != NULL);
14   size_t len = end - start;
15   char b64[len + 1];
16   memcpy(b64, start, len);
17   b64[len] = '\0';
18   // [...]
19 }
```

Fig. 13: Previously unknown vulnerability found using the first traversal.

an attacker-controlled integer are common sources for buffer overflows, and hence we select the query.

Table V shows the two functions returned by the query, both of which are vulnerable. In particular, the function `Encode` in the source file `modules/codec/dirac.c` as shown in Figure 14 causes a buffer overflow: the 32 bit variable `len` is initialized by the attacker-controlled source `GetDWBE` on line 5 and used in the allocation on line 7. Unfortunately, the fixed value `sizeof(eos)` is added to `len` directly before allocation, causing an integer overflow. In effect, too little memory is allocated for the buffer `p_extra`. Finally, on line 10, `len` bytes are copied into the undersized buffer causing an overflow.

In summary, we identified 5 previously unknown vulnerabilities, that can possible be exploited to execute arbitrary code.

```
static block_t *Encode(encoder_t *p_enc, picture_t *p_pic)
{
 if( !p_enc->fmt_out.p_extra ) {
   // [...]
   uint32_t len = GetDWBE( p_block->p_buffer + 5 );
   // [...]
   p_enc->fmt_out.p_extra = malloc( len + sizeof(eos) );
   if( !p_enc->fmt_out.p_extra )
     return NULL;
   memcpy( p_enc->fmt_out.p_extra, p_block->p_buffer, len);
   // [...]
 }
}
```

Fig. 14: Previously unknown vulnerability found using the second traversal.

Moreover, we achieved this by selecting just two promising automatically generated queries, illustrating the practical merits of our method as a tool for security analysts who review code for vulnerabilities.

VI. Limitations

The discovery of previously unknown vulnerabilities through automatically inferred search patterns demonstrates the merits of our method. Nevertheless, there exist certain limitations that we discuss in the following.

First and as a consequence of our inference setup, if a major part of the code base lacks a proper sanitization, our method is unable to identify corresponding sanitization rules from the data flow and thus fails to generate accurate search patterns. Fortunately, this limitation does not apply to mature software projects that make extensive use of sanitization when processing user-controlled data.

Second, our method assumes that data is only passed from callers to the callee via function arguments and return values. Shared resources, such as global variables or shared memory, are not modeled by our method. As a consequence, we are not able to describe taint-style vulnerabilities where an attacker propagates data across these resources to a sink. Modifying the interprocedural code property graph to account for this type of data flow seems involved but possible. We leave this modification as an extension for future work.

Third, the work discussed so far only shows the applicability of our method for the identification of vulnerabilities typical for C code, such as invalid memory accesses. While in principle, our method should be applicable to several other vulnerability types, and in particular, typical Web application flaws, this remains to be shown. In particular, adapting our method to a different language requires careful handling of language-specific properties.

Finally, the control flow of a software is not fully recovered by our method. In particular, dynamic calls are currently not resolved. Similarly, our method is not able to describe vulnerabilities rooted in concurrent execution of functions, such as many use-after-free security flaws. This limitation is not trivial to address and possibly benefits from coupling code property graphs with techniques for dynamic analysis, such as dynamic taint tracking or symbolic execution.

VII. Related Work

The development of methods for finding vulnerabilities in software is long-standing topic in security research that spans a wide range of approaches and techniques. For our discussion of related work, we focus on approaches that also aim at assisting a security expert during auditing of software instead of replacing her.

a) Methods based on query languages and annotations: Closely related to our work are approaches that enable an analyst to search for vulnerabilities using query languages or annotations. For example, the methods by Martin et al. [35] and Lam et al. [30] both employ descriptive query languages for modeling code and finding software defects. Similarly, Vanegue et al. [56] experiment with extended static checking [15] as part of the HAVOC tool and test its performance at a large code audit. Moreover, several approaches for the discovery of information-flow vulnerabilities based on security type systems have been presented [see 21, 47, 50]. In particular, the *Jif* compiler [38, 39] performs type checking to allow security policies to be enforced for an annotated version of Java. Moreover, Jif implements a type inference algorithm to reduce the number of user-defined annotations required.

Finally, Evans and Larochelle [13] use annotations for C as a means for finding vulnerable patterns in code. While our approach shares a similar motivation, it differs in that it automatically infers search patterns and thus the analyst only needs to define a set of security-sensitive sinks to start auditing an unknown code base.

b) Inferring programming patterns and specifications: Manual analysis of code is a tedious and time-consuming task. As a remedy, several methods have been proposed that make use of statistical methods, machine learning, and data mining techniques for accelerating this process. To this end, several methods automatically infer programming patterns [e.g., 19, 32, 58] and security specifications [e.g., 28, 34, 54], from code, revision histories [33], and preconditions of APIs [e.g., 7, 41, 55]. A related strain of research has followed a more principled approach by modeling and inferring security policies [e.g., 6, 36, 52, 57] for discovering information-flow vulnerabilities. Similar to our method, many of these approaches are based on syntax trees and code slices as well as representations that combine syntax, control flow, and data-dependence relationships [e.g., 27, 29].

Engler et al. [12] are among the first to point out that defects in source code can often be linked to violations of implicitly introduced system-specific programming patterns. They present an approach to automatically tailor user-supplied rule templates to specific systems and demonstrate its ability to identify defects in system code. Closely related to this work, Kremenek et al. [28] go one step further by showing that an approach based on factor graphs allows different sources of evidence to be combined automatically to generate specifications for violation detectors.

More closely related to vulnerability discovery, Livshits et al. [34] present Merlin, a method based on factor graphs that infers information flow specifications from Web applications for the Microsoft .NET framework. An important limitation of Merlin is that it only models the flow of information between functions, and hence, sources, sanitizers and sinks are always assumed to be calls to functions. While for typical Web application vulnerabilities, this assumption holds in many cases, missing bounds checks for vulnerabilities such as buffer overflows or null pointer checks cannot be detected in this way. In contrast, our method is well suited to encode these checks as sanitizers are derived from arbitrary statements, allowing patterns in declarations and conditions to be modeled (see Section V). Similarly, Yamaguchi et al. [62] present Chucky, an approach to the detection of missing checks that is also capable of dealing with sanitizers given by arbitrary condi-

tions. Unfortunately, the approach is opaque to the practitioner and thus a control or refinement of the detection process is impossible. In contrast to both Merlin and Chucky, sources, sanitizers, and sinks are expressed as regular expressions as part of traversals, making it easy for the analyst to adapt them to further improve the specification. Finally, several authors employ similarity measures to determine vulnerabilities similar to a known vulnerability [17, 24, 42, 61].

c) Methods based on dynamic analysis: A considerable body of research has focused on exploring dynamic code analysis for vulnerability discovery. Most notably are black-box fuzzing [e.g., 43, 53] and white-box fuzzing techniques [e.g., 18, 20, 59]. These approaches are orthogonal to our work, as they explore the data flow in source-sink systems at run-time. Although not specifically designed to assist a human analyst, white-box fuzzing might complement our method and help to explore which parts of the code are reachable by attackers to further narrow in on vulnerabilities.

VIII. Conclusion

The discovery of unknown vulnerabilities in software is a challenging problem, which usually requires a considerable amount of manual auditing and analysis work. While our method cannot generally eliminate this effort, the automatic inference of search patterns significantly accelerates the analysis of large code bases. With the help of these patterns, a practitioner can focus her analysis to relevant code regions and identify taint-style vulnerabilities more easily. Our evaluation shows that the amount of code to audit reduces by 94.9% on average and even further in the case of the "Heartbleed" vulnerability, showing that automatically generated search patterns can precisely model taint-style vulnerabilities.

Our work also demonstrates that the interplay of exact methods, such as static program analysis, with rather fuzzy approaches, such as machine learning techniques, provides fruitful ground for vulnerability discovery. While exact approaches offer a rich view on the characteristics of software, the sheer complexity of this view is hardly graspable by a human analyst. Fuzzy methods can help to filter this view—in our setting by search patterns—and thus guide a practitioner when auditing code for vulnerabilities

Reporting of Vulnerabilities

We have worked with the vendor to fix all vulnerabilities identified as part of our research. Upcoming versions should no longer contain these flaws.

Acknowledgments

We acknowledge funding from DFG under the project DEVIL (RI 2469/1-1). We would also like to thank Google, and in particular our sponsor Tim Kornau, for supporting our work via a Google Faculty Research Award. Finally, we thank our shepherd Andrei Sabelfeld and the anonymous reviewers for their valuable feedback.

References

[1] The Heartbleed Bug, http://heartbleed.com/, 2014.

[2] The Shellshock Vulnerability. http://shellshockvuln.com/, 2014.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[4] M. Anderberg. *Cluster Analysis for Applications*. Academic Press, Inc., New York, NY, USA, 1973.

[5] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder's Handbook: Discovering and exploiting security holes*. John Wiley & Sons, 2011.

[6] M. Backes, B. Kopf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *Proc. of IEEE Symposium on Security and Privacy*, 2009.

[7] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Transactions on Software Engineering*, 34 (5):579–596, 2008.

[8] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001.

[9] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2006.

[10] J. Dahse and T. Holz. Simulation of built-in PHP features for precise static code analysis. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2014.

[11] M. Dowd, J. McDonald, and J. Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[13] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19 (1):42–51, 2002.

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1987.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM Sigplan Notices*, volume 37, pages 234–245, 2002.

[16] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *Proc. of the ACM workshop on Artificial intelligence and security*, 2013.

[17] F. Gauthier, T. Lavoie, and E. Merlo. Uncovering access control weaknesses and flaws with security-discordant software clones. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2013.

[18] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE:

Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[19] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[20] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proc. of the USENIX Security Symposium*, 2013.

[21] N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proc. of the ACM Symposium on Principles of programming languages (POPL)*, 1998.

[22] S. Hido and H. Kashima. A linear-time graph kernel. In *Proc. of the IEEE International Conference on Data Mining (ICDM)*, 2009.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proc. of the ACM International Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, 1988.

[24] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[25] M. A. Jaro. Advances in record linkage methodology as applied to the 1985 census of Tampa Florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[26] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. of IEEE Symposium on Security and Privacy*, 2006.

[27] D. A. Kinloch and M. Munro. Understanding C programs using the combined C graph representation. In *Proc. of the International Conference on Software Maintenance (ICSM)*, 1994.

[28] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proc. of the Symposium on Operating Systems Design and Implementation*, 2006.

[29] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11): 661–675, 1998.

[30] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proc. of Symposium on Principles of Database Systems*, 2005.

[31] S. Lekies, B. Stock, and M. Johns. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[32] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of European Software Engineering Conference (ESEC)*, pages 306–315, 2005.

[33] B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Proc. of European Software Engineering Conference (ESEC)*, pages 296–305, 2005.

[34] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proc. of the ACM International Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[35] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: Program Query Language. In *Proc. of ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2005.

[36] I. Mastroeni and A. Banerjee. Modelling declassification policies using abstract domain completeness. *Mathematical Structures in Computer Science*, 21(06):1253–1299, 2011.

[37] D. Muellner. Fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python. *Journal of Statistical Software*, 53(9):1–18, 2013.

[38] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proc. of the ACM Symposium on Principles of programming languages (POPL)*, 1999.

[39] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2001.

[40] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. of IEEE Symposium on Security and Privacy*, 2005.

[41] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of APIs in large-scale code corpus. In *Proc. of the ACM International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[42] J. Pewny, F. Schuster, C. Rossow, L. Bernhard, and T. Holz. Leveraging semantic signatures for bug search in binary programs. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2014.

[43] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proc. of the USENIX Security Symposium*, 2014.

[44] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 1998.

[45] K. Rieck, C. Wressnegger, and A. Bikadorov. Sally: A tool for embedding strings in vector spaces. *Journal of Machine Learning Research (JMLR)*, 13(Nov):3247–3251, Nov. 2012.

[46] M. A. Rodriguez and P. Neubauer. The graph traversal pattern. *Graph Data Management: Techniques and Applications*, 2011.

[47] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[48] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX:

Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2010.

[49] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of IEEE Symposium on Security and Privacy*, 2010.

[50] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. of the USENIX Security Symposium*, 2001.

[51] S. Son, K. S. McKinley, and V. Shmatikov. RoleCast: Finding missing security checks when you do not know what checks are. In *Proc. of ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

[52] V. Srivastava, M. D. Bond, K. S. Mckinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple API implementations. In *Proc. of the ACM International Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[53] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[54] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *Proc. of the USENIX Security Symposium*, 2008.

[55] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proc. of the International Conference on Automated Software Engineering (ASE)*, pages 283–294, 2009.

[56] J. Vanegue, L. Bloomberg, and S. K. Lahiri. Towards practical reactive security audit using extended static checkers. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[57] J. A. Vaughan and S. Chong. Inference of expressive declassification policies. In *Proc. of IEEE Symposium on Security and Privacy*, 2011.

[58] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger. JIGSAW: Protecting resource access by inferring programmer expectations. In *Proc. of the USENIX Security Symposium*, 2014.

[59] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2009.

[60] M. Weiser. Program slicing. In *Proc. of International Conference on Software Engineering*, 1981.

[61] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[62] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[63] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of IEEE Symposium on Security and Privacy*, 2014.