An User Configurable Clang Static Analyzer Taint Checker

Marcelo Arroyo, Francisco Chiotta and Francisco Bavera
Departamento de Computación, FCEFQyN
Universidad Nacional de Río Cuarto
Córdoba, Argentina

***Abstract*—In this paper, we describe the development and usage of clang static analyzer checker for detecting tainted data in C,C++ and Objective C source programs. The checker is user configurable, so it can be used to check tainted data for any user provided API. It also include subsets of C/C++ APIs commonly used as memory and string handling and file input-output.**

Taint checking is a widely used technique as source code review tools to detect possible security vulnerabilities for attacks as code injection and buffer overflows.

We describe the clang static analyzer architecture, the taint checker design considerations, some implementation details and some test cases to show the capability for detecting security vulnerabilities as the *hearthbleed* in a real and big open source project such as *OpenSSL*.

***Index Terms*—Security, taint checking, static analysis, symbolic execution.**

I. Introduction

In the recent years, compilers have evolved to more open architectures, providing and exposing some of its internal components to programmers, enabling the customization of the compilation process. By exposing the stages of compilation and internal program representation data structures, as abstract syntax trees (AST), control flow graphs (CFG), and including plug-ins support, modern compilers enable developers to include some extra functionality in some stages of compilation process.

The `CLANG-LLVM`[2][3] project was born with the goal to produce a set of reusable components (in contrast with monolitic architecture of previous versions of popular GCC[1]) for building a complete compiling suite. Actually, the `clang` compiler is used for Apple as its main compiling technology for MAC OS X and IOS platforms.

Taint checking or *taint analysis* is a feature found in some programming languages such as `Perl` and `Ruby`, designed to increase security against malicious attacks such as code injection (sql, javascript and others) and buffer overflows.

The `Open Web Application Security Project`[19] define a taxonomy of security vulnerabilities. The category *Input Validation Vulnerability* include buffer overflows, format and encoding strings and others. This group of vulnerabilities can be detected by static or dynamic taint analysis.

The main idea of taint analysis is that any variable that can be modified (direct or indirectly) by user inputs, becomes a security risk. A variable which can store external or unsafe data is called a *tainted variable*.

Consider the following (naive) C++ code fragment in figure 1:

```
bool authenticate(char* user,char* pass)
{
  string query = "select * from USER where
  user='"+user+"' and pass='"+password+"'";
  Result r = execsql(conn, query);
  if (r.size()==0)
    return false; // accesss denied
  return true;
}
```

Fig. 1. An example of SQL code injection vulnerability.

A function call as `authenticate("any", "'any' or true;");` makes `authenticate()` function always return true.

This is a very simple example of a well known vulnerability which enable a malicious user to inject SQL code. In similar scenarios a program could be vulnerable to stack or buffer overflow attacks.

978-1-5090-3339-3/16/$31.00 ©2016 IEEE

These kind of vulnerabilities can be fixed by filtering program inputs and just to accept restricted (safe) patterns. In this example, the formal parameters `user` and `pass` should be considered *tainted* because their contents depends on external inputs. The local variable `query` is tainted too (because its value depends on `user` and `pass`) at program point of call to `execsql()` function. The `execsql()` function is an example of a *sink* of tainted data.

In more complex and long programs, this kind of vulnerabilities can be very hard to detect by simple human code inspection[1], so automatic discovering of tainted data becomes an important tool for improving software security.

Informally, taint analysis is a security mechanism for the following security policy:

Data from untrusted sources is tainted and it is not allowed to reach some specific critical functions (sinks) without passing by security data constraints validators or sanitizers.

In general, two different approaches have been explored with the aim of providing taint analysis, static and dynamic, each associated with compile-time and run-time systems. Run-time solutions take a different approach by using the labels as an extra property of the object and tracking their propagation as the objects are involved in computation.

In the compile-time approach, applications are written in specially designed programming languages in which special annotations are used to attach security labels and constraints to the objects in the program. At compile time, the compiler uses these extra labels to ensure authorized flows of information.

These compile-time checks can be viewed as an augmentation of type checking [29], [30]. In type-based analysis, types are extended so that they include security information. The type checker applies the typing rules to track security information, and warns on a rule violation.

Static taint checking is just a special case of static analysis based on information flow control[6][7] and noninterference[8][9]. A program is *information flow secure* if an attacker cannot execute critical computations on data manipulated from untrusted sources.

Properties related to information flow are *integrity* and *confidentiality*.

Confidentiality prevents data flow from flowing to undesired destinations. Integrity ensures data was not altered in a not authorized way. Taint analysis could be considered as a form of integrity analysis. Integrity and confidentiality were long considered dual to each other [28]. There is an important amount of research effort on checking integrity and confidentiality properties using information flow analysis[27].

The starting point in secure information flow analysis is the classification of program variables into different integrity security levels. Following standard practices, we'll only use two security levels: tainted information (*tainted*) and untainted information (*untainted*). An insecure flow of information occurs when the values of tainted variables (untrusted data, e.g. user inputs) reach and influence untainted parts of the system (vulnerable functions).

For example, we would allow flows from $tainted$ to $tainted$, from $untainted$ to $untainted$, and from $untainted$ to $tainted$, but we would disallow flows from $tainted$ to $untainted$ (except for filters).

Noninterference is a correctness property of information flow analysis. This property intuitively means that tainted values do not interfere with the untainted operations.

Tools based on static analysis are in general fully automated. This approach have the advantage that they are very easy to use because programmers doesn't need to have a fully understanding of the internal aspects or details of how the tool works.

Obviously, static taint checking is an undecidable property[2] in the general case, so we can develop just a conservative analysis. In practice, taint analysis have been used successfully to detect insecure data flows in real projects.

The main disadvantage with static analysis techniques is that it could be incomplete (can miss some real

[1]For example, the *hearthbleed* vulnerability in OpenSSL is present in several mini-releases.

[2]Is simple to show that taint analysis is equivalent to the program halting problem.

problems) and unsound (generate false warnings), also known as *false positives*.

Completeness can be achieved by means of a sufficient conservative approach, but it could increase the number of false positives. Reducing the number of false positives while maintaining completeness is the main challenge in the development of static analysis algorithms.

In this paper, we describe the development of an user configurable taint checker plug-in for the *clang static analyzer*[13] as our main contribution. Also, we include a description of the *clang static analyzer* architecture and how to write custom checkers (analyzer plug-ins). The *clang static analyzer* architecture and how to write checkers is poorly documented and actually doesn't exist a specific paper describing its internals. So, we think that it is a secondary contribution.

This papers is organized as follows: In the next section, describe some related work and similar tools. Then, we describe the basic concepts about taint analysis and typestate systems. In section IV we describe some details about the clang static analyzer core architecture and checkers structure. Next, we describe the design and some implementation details of the taint checker developed. Finally, we show some experimental results, conclusions and future work in the area of program source based security and static analysis into the `clang-llvm` framework.

II. Related work

Static analysis techniques include numerous approaches. Lightweight static analysis is based in existing compilers technology. Type systems and static type checking are used for semantic analysis to reject not meaningful programs.

More powerful static analysis include property checking and program verification. In general this techniques includes the generation of *verification conditions* to be proved with specialized semi-automatic theorem provers. In the general case this techniques require user intervention for proving some kind of properties.

In this work, we focus in lightweight static analysis techniques that can be applied to big software projects as a fully automatic process.

Most approaches for static analysis are based in abstract interpretation techniques[10]. Specialized type systems[11] (for security, concurrency, memory safety, and others) have been developed to increase the expressiveness of the verification process. In particular, type systems for security have been used to describe (sometimes with program type annotations) confidentiality policies to enforce some kind of *noninterference*: a property that requires high-level security information do not affect low-level security computations.

By extending traditional type systems, we can mechanically to check some complex properties and to detect new kind of bugs. Typestate systems[12] extend a traditional type system adding state changes on a given set of operations. Typestate is useful to checking application programming interfaces (APIs), including implicit resource usage protocols such as explicit memory handling and file and network APIs.

Types and effect systems[22] allows to add extra information to traditional types and to check *computation side effects*. Some application examples are type systems for concurrency, safe memory management, energy consumption models and others.

Static and dynamic analysis has been widely used in code based security. Taint checking[3] is very useful to finding security vulnerabilities. Some specific tainted checking tools as `BitBlaze`[17] and `TaintCheck`[18] performs dynamic taint analysis. Dynamic approaches are out of scope in this work because we are interested in static analysis techniques.

Some static analysis tools are based on well known compilers technology, by constructing an internal program representation, such as *Abstract Syntax Tree (AST)*, *Control Flow Graphs (CFGs)* and *Call Graphs (CG)* and then applying data and control flow algorithms. By using typestate systems the the taint analysis problem is reduced to a graph reachability problem.

`Coverity Code Advisor`[14] is a well known commercial and enterprise widely used tool that performs static analysis for C, C++, Java and other programming languages for finding bugs as bounds

[3]Also known as *user-input dependency analysis.*

checking, API usage conformance and security vulnerabilities including taint checking and other specialized checks. It support programmer defined configurations (XML files) for building customized checks.

`Splint`[15] is an open source tool also based in traditional compilers technology. It supports program annotations to check some program properties.

`Lint`-like tools[5], which performs classical static analysis, almost exist for any existing programming language.

`Frama-C`[25][26] supports only the ANSI C language (not C++ and Objective-C) and is based in an general and extensible abstract interpretation framework. `Frama-C` is written in Ocaml and provides a specification language (`ACDL`) for describing user-defined ckecks. Also, Frama-C supports plug-ins to implement more complex analyzers. Some plug-ins (shipped with the standard distribution) focus on computing metrics, verification for annotated programs with linear temporal logic specifications, weakest precondition formulas, and value analysis. Actually, `Frama-C` doesn't provide a tainted checker but it could be expressed in the `ACDL` language or by writing a specific plug-in.

There are many tools based on static analysis techniques for other programming languages as `FindBugs`[16].

III. Taint Analysis and typestate systems

In this section, we define static taint analysis in a more precise way. As we said before, taint analysis is a special case of data flow analysis.

Definition 1: Data **flows** from x to y, written as $x \rightarrow y$, when data referenced by x is copied to y.

So, any operation causing a copy, as assignments or parameter passing, cause a data flow.

Definition 2: A variable v containing data from unknown or untrusted sources is **tainted**.

Definition 3: An operation *op*, returning a result r from a **tainted** variable v, written as $v \rightarrow op(r)$, is considered a **taint operator**. Also, r is considered tainted too.

In this way, we can see that a *taint operator* op is transitive: if $x \rightarrow op(y)$ and $y \rightarrow op(z)$, then $x \rightarrow z$.

Taint operators can *generate* (called ***sources***) or *propagate* tainted data (***propagators***).

However, not all operators taking *tainted data* should be considered source or propagators. Some operators can generate safe (untainted) data from tainted data. They are known as ***sanitizers*** or ***filters***. A *filter* convert (or propagate previous checking) tainted data into untainted data.

Static taint analysis can be implemented using *typestate* systems[12]. *Typestates* are capable of representing behavioral type refinements by associating states to variables of a given type at different program points and has been used as lightweigth contracts in object-oriented languages and for API usage conformance. Some common examples are resource usage checking with *get-use-free* semantics such as files, network sockets and dynamic memory allocation. It can be applied to other domains too, such as variable initialization before use checking or taint analysis.

Taint analysis can be seen as a generic API definition with three sets of operations:

- $Sources$ is the set of operations that can generate tainted data.
- $Propagators$ is the set of operations that propagate or transform tainted data in tainted data. Any programming language primitive operation as assignments or parameter passing are considered *propagators*. Aliasing should be considered if the programming language uses *references* or *pointers*. Any copy operation like assignments or parameter passing are propagators.
- $Filters$ is the set of operations that checks if data is safe or generate safe data from tainted data.
- $Sinks$ is the set of (critical) operations that use (consume) data.

A typestate system define the operations (expressions and function arguments and results) and how they change state. A such system can be described with typing rules or with finite state machines or *labeled Transition Systems* (LTS).

Definition 4: A (general) typestate system is a labeled transition system $LTS =< \tau, O, Q, T, E >$ with

- τ is the set of types of values and variables of interest.
- Σ is the set of *operations* taking arguments and/or returning values of type τ. Each $op \in Sigma$ includes information about its signature (arguments and result types).
- Q is the set of *typestates* including $\bot$ (the lower typestate).
- $T : \Sigma \times Q \to Q$ is the *transition relation*.
- E is the set of *error* typestates.

Storm and Yemini in [12] require that exist a partial order on states to be a *semi-lattice* with a *least element* ($\bot$). This is necessary because in the context of classical static analysis algorithms, a *path merging*[4] some operation on *typestates* is required: $ts\_merge(\_, \bot) = ts\_merge(\bot, \_) = \bot$.

For taint analysis, we are interested in states $\{\bot, tainted, untainted\}$ for specific variables or values at different program points, with the ordering $\bot \prec tainted \prec untainted$.

Definition 5: A **typestate system for taint analysis** is a $LTS_t =< \tau, O, Q, S, T, E >$ where

- τ is the set of types of values and variables of interest.
- $\Sigma = S_o \cup P \cup F \cup S_i$, is the set of operation's signatures, where S_o is the set of *sources*, P is the set of *propagators* F is the set of *filters* and S_i is the set of *sinks*.
- $S = \{tainted, untainted, error\}$, where $tainted$ represent the (*lower*) *tainted* typestate ($\bot$).
- T is the typestate transition relation defined as:
  $T(\_, op) = tainted$ if $op \in S_o$
  $T(s, op) = s$ if $op \in P$ ($s \in Q$)
  $T(\_, op) = untainted$ if $op \in F$.
  $T(untainted, op) = error$ if $op \in S_i$
  $T(error, \_) = error$
- $E = \{error\}$

The typestate system described above can be seen as a transition system as shown in figure 2.

In the context of symbolic execution[5], sometimes we can relax the typestate ordering requeriment because we don't have to do *path merging*, as required by classical static analysis algorithms. Symbolic execution consider (*walk*) different paths at different times.

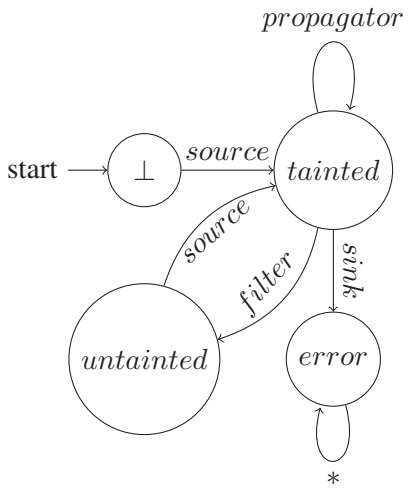

Fig. 2. State transition system for taint analysis.

A taint static checker algorithm requires to compute typestates at each program point. As said before, algorithms are based on building a *control flow graph (CFG)* for each function in the program. The main idea behind a static tainted checker algorithm is to keep track typestates at each program point (before/exit of each CFG node).

The clang static analyzer core provides a symbolic execution machinery. The symbolic execution engine walks the program abstract syntax tree (AST)[6] building the CFG making flow-sensitive analysis. On each CFG node, the core engine calls to core and other registered checkers for each program point[7].

The framework enable us to write specific checkers working collaboratively with others.

In our general clang analyzer tainted checker, the sets S_o, P, F and S_i are user defined in a XML file. For each operation, user have to define what arguments are involved and how (input or output). The checker handles automatically all *copy* and aliasing operations for the C/C++/Objective C programming languages using contextual information provided by the core

[4]Path merging require computing typestate in a joining branch program point.

[5]The clang static analyzer core performs symbolic execution.

[6]The AST is built by the clang parser.

[7]A CFG node as a function call have a pre-call and a post-call program points

system and the symbolic execution engine.

IV. The Clang static analyzer

The `Low Level Virtual Machine (LLVM)` [2] project was born with the aim of providing a set of reusable components for a back-end compilation suite, including a virtual low-level general instruction set representation (IR), code generators for many CPU targets, optimizers and linker[8].

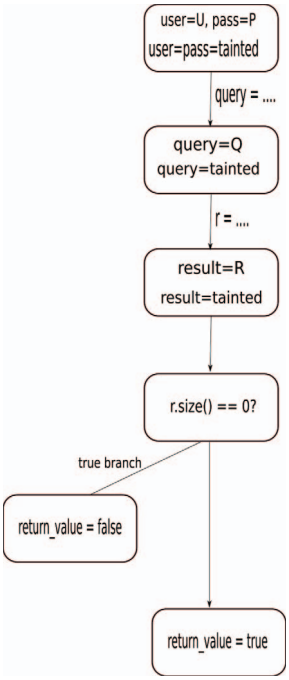

Fig. 3. Graph of reachable states

The `Clang` project provides the front-end (parser and related tools) for C based languages (`C/C++/Objective C`) for the `LLVM` infrastructure.

Following the same design principles of `LLVM`, `CLANG` is built as a set of reusable components (libraries) that can be used for development of other tools. For example, the integrated parser and semantic libraries (AST building) have been used for creating tools as `OCLint`[4] (for automatic code review process) and others.

The *static analyzer* is a component included in `clang`. It is based on symbolic execution with an open architecture. The core engine performs path-sensitive exploration of the source program and relies on a set of checkers which implements the logic for detecting specific bugs.

It is based around the concepts of precise inter-procedural data-flow analysis via graph reachability[20] and a **C** based programming languages memory model[21].

The core engine builds a *graph of reachable states*[9] for each function. The figure 3 shows the graph corresponding to program of figure 1.

The values denoted with upper case letters represents symbolic values. A taint checker could to compute states for each symbol (variable) at each program point for symbols as shown.

Checking if tainted data can reach the `r` variable or `execsql()` function (sink) can be reduced to a graph reachability problem. In this example, the checker will generate a warning because there is a such path.

The clang static analyzer define the following components:

- ***Core engine***: walks the AST, build the CFG and generate events which are notified to core and other registered checkers (by a callbacks mechanism).
- ***State manager***: handle program states.
- ***Constraint Manager***: store and solve path conditions.
- ***Store Manager***: store of symbolic values and memory regions.

Explored (symbolic) execution paths are represented with an `ExplodedGraph` object. Each node of the graph is an `ExplodedNode` object which contains two main properties:

- `ProgramPoint`: represents the program location (entry/exit CFG's node). It contains information about the exact symbolic execution location (before/after a statement, entering/returning to/from a call, etc) and the symbolic stack frames.
- `ProgramState`: represents the abstract state of the program. It consist of:
  - `Environment`: a mapping from source code expressions to values.
  - `Store`: a mapping from memory locations to symbolic values. It represents a instance of the symbolic memory model.
  - `GenericDataMap`: Constraints on symbolic values.

[8]It is also known as a *toolchain*.

[9]Which is a CFG augmented with checkers state

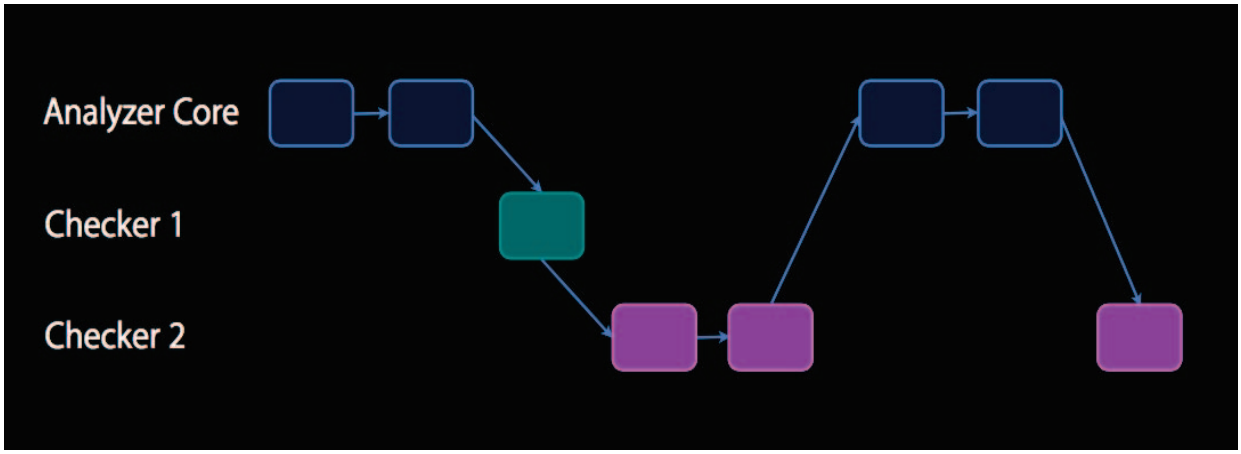

Fig. 4. Checkers program states.

On each program point of the source program, the core engine notifies each registered checker (a call to a specific member function) giving to them an opportunity that each checker to modify the state or to report a potential bug. Actually, checkers are (stateless) visitors.

The figure 4 shows a schema of how the core engine works together with checkers.

The symbolic execution engine uses a simple and fast constraint solver[10] to prune unfeasible paths (when it is possible). The constraint solver helps in prunning some unfeasible paths when a given set of constraints are unsatisfiable.

For example, in the following program fragment, the path $< 1, 2, 3, 4 >$ is unfeasible, because the constraint set $\{x < y, x == y\}$ is unsatisfiable.

```
if (x < y)      // 1
   z = y;       // 2
if (y == x)     // 3
   z = x;       // 4
```

Sometimes the constraint solver can't solve if a given constraint set is satisfiable or not[11]. In these cases, the symbolic execution engine have to take both paths, in order, assuming constraints as `true`.

Symbolic execution bind to variables (and temporary values), at each program point, concrete (known) or symbolic values, depending on expressions complexity and the context.

Symbolic values are represented by `SVal` (symbol) objects. A `SVal` can represent a concrete value of a given type (the `int` 0, for example), symbolic (constrained) values or memory locations (regions) or an `unknown` value (when the constraint solver can't track values). For example, when a `SVal` object represent a concrete integer, say 42, it is actually an instance of `ConcreteInt`. In cases when `SVal` represents a symbolic value it is an instance of `SymExpr`, which represents an inmutable without a specific value but it can have associated constraints (for example a positive integer).

Abstract memory is represented with `MemRegion` objects and it similar to symbol. Regions can layer on top of other regions, providing a layered approach to representing memory. For example, a struct object on the stack might be represented by a `VarRegion`, but a `FieldRegion` which is a subregion of the `VarRegion` could be used to represent the memory associated with a specific field of that object.

This layered approach allow efficient representation of values. For example, if a compound object is initialized with zero, is only necessary to bind the value to the top-level `MemRegion`.

Symbolic memory regions are represented by `SymbolicRegion` objects, which are memory regions associated to a symbol. It is used for modeling pointer and reference variables.

[10]Based on interval arithmetic.

[11]In general, a constraint solver result is a set of values (or value ranges) when constrains are satisfiable, the empty set when constraints are unsatisfiable and *I don't know* otherwise.

The machinery of clang analyzer allow us to write checkers to implement *type and state* or more general *type and effect* systems[22] to detect bugs.

The clang static analizer provides a command-line tool (`scan-build`) which generate HTML reports. Also it can be integrated with XCode, presenting reports as shown in figure 5.

In the next section we show our taint checker implementation details.

V. An user configurable taint checker

Checkers are plug-ins in the clang static analyzer framework. A checker is just a stateless registered visitor.

A checker inherit from the `Checker` variadic template class. The `Checker` template parameters describe the type of events that the checker is interested in processing. For each event type requested, a corresponding callback function must be defined in the checker class.

In particular, our taint checker is interested in processing the following events:

- `checkPreStatement<CallExpr>`: dispatched before processing a function call.
- `checkPostStatement<CallExpr>`: dispatched after a processing a function call.
- `checkBind`: dispatched on each binding of a value to a location.
- `checkDeadSymbols`: dispatched when a symbol becomes dead.

The figure V show the general checker structure.

Checkers often need to keep track of information specific to the checks they perform. However, since checkers have no guarantee about the order in which the program will be explored, or even that all possible paths will be explored, this state information cannot be kept within individual checkers. Therefore, if checkers need to store custom information, they need to add new categories of data to the ProgramState. Some macros are provided to include extra information to `ProgramState`:

- `REGISTER_TRAIT_WITH_PROGRAMSTATE`
- `REGISTER_LIST_WITH_PROGRAMSTATE`
- `REGISTER_SET_WITH_PROGRAMSTATE`
- `REGISTER_LIST_WITH_PROGRAMSTATE`

The first container is used to store single values. Other container templates are used to implement more complex state containters.

For example, a common case is the need to track data associated with a symbolic expression; a map type is the most logical way to implement this. The key for this map will be a pointer to a symbolic expression (SymbolRef). If the data type to be associated with the symbolic expression is an integer, then the custom category of state information would be declared as

```
REGISTER_MAP_WITH_PROGRAMSTATE(
ExampleDataType, SymbolRef, int)
```

These data structures provides methods for getting, setting and finding values.

Program states are *immutable*. A new state can be provided to analyzer core by calling `CheckerContext::addTransition` function.

In a case that a checker require analysis can't continue (for example, when a severe error was detected), the current state should be transitioned into a so-called *sink node*[12] by calling to the `CheckerContext::addSink` function.

When a checker detects an error in the analized code, it need to report it to analizer core so that it can be displayed. There are two classes to build an error report: `BugType` and `BugReport`. The former class, represents the bug type (name and category). The `BugReport` class represents a specific bug instance and its constructor takes three parameters:

1) The bug type: an instance of `BugType`.
2) A short descritptive string.
3) The context in which the bug occurred. This includes both the location of the bug in the program and the program's state where the location is reached. Both data is then encapsulated in an `ExplodedNode`.

In order to obtain the correct ExplodedNode, a decision must be made as to whether or not analysis can continue along the current path. This decision is based on whether the detected bug is one that would prevent the program under analysis from continuing. For example, leaking of a resource should not stop analysis, as the program can continue to run after the leak. Dereferencing a null pointer, on the other hand,

[12]Do not confuse with taint checking *sink* operations.

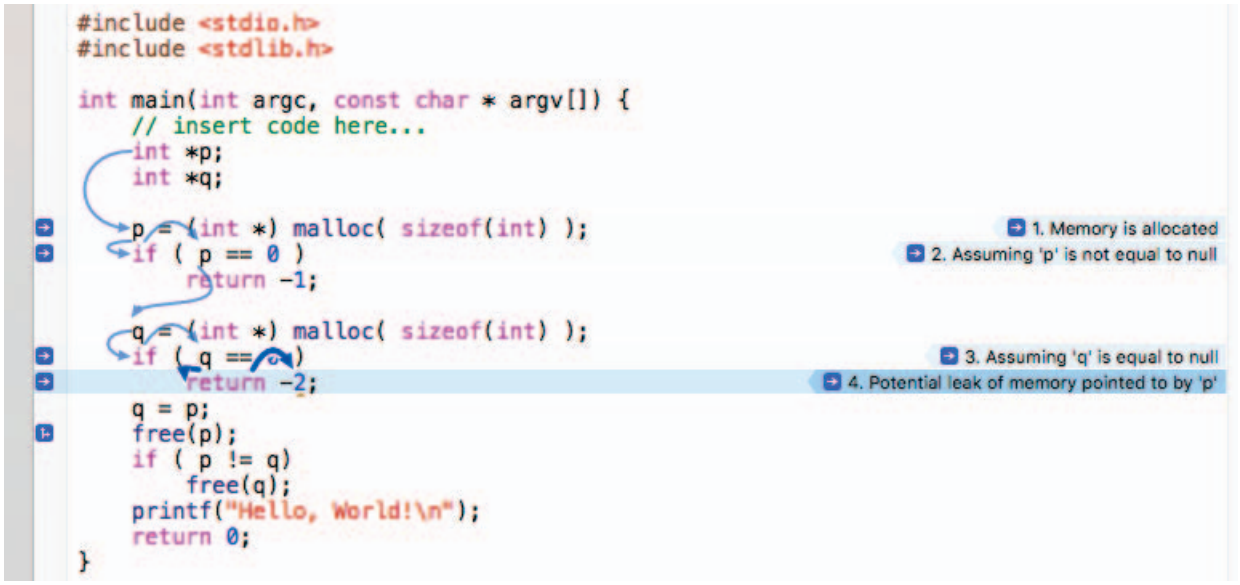

Fig. 5. Clang static analyzer integrated in Xcode.

```
...
class CustomTaintChecker :
  public Checker<check::PostStmt<CallExpr>,
                 check::PreStmt<CallExpr>,
                 check::Bind>
{
public:

  void checkPostStmt(const CallExpr *CE,
                     CheckerContext &C)
                    const;
  void checkPreStmt(const CallExpr *CE,
                     CheckerContext &C)
                    const;
  void checkBind(SVal Loc, SVal Val,
                 const Stmt *S,
                 CheckerContext &) const;
  ...
};
```

Fig. 6. The taint checker structure.

should stop analysis, as there is no way for the program to meaningfully continue after such an error.

If analysis can continue, then the most recent `ExplodedNode` generated by the checker can be passed to the `BugReport` constructor without additional modification. This `ExplodedNode` will be the one returned by the most recent call to `CheckerContext::addTransition()`. If no transition has been performed during the current callback, the checker should call `CheckerContext::addTransition()` to get the current `ExplodedNode` state for bugs reporting.

If analysis can not continue, then the current state should be transitioned into a so-called sink node, a node from which no further analysis will be performed. This is done by calling the `CheckerContext::generateSink()` function; this function is the same as the `addTransition()` function, but marks the state as a sink node. Like `addTransition()`, this returns an `ExplodedNode` with the updated state, which can then be passed to the `BugReport` constructor.

After a `BugReport` is created, it should be passed to the analyzer core by calling to the `CheckerContext::emitReport()` function.

Our checker extends and refine an existing taint checker already included in the analyzer. This checker includes checking for hardcoded APIs as UNIX system calls, some C standard library functions as `memcpy`, `strcpy`, `...` and some functions of the standard file API.

Our extended taint checker additionally includes:

- Handling a configuration file for user API description of sources, filters, propagators and sinks.

- Take into account the manipulation of composite objects: When an object is passed as parameter to a function, it could access or update internal fields, which could be (or should be marked as) tainted. To address this problem our checker inspect inside compound objects (structs and classes).

The configuration file is based in XML and allow developers to define sources, propagators, filters and sinks in a precise way. For each operation we have to define its name and which argument (or result) is affected.

Our taint checker is available at `https://github.com/franchiotta/taintchecker`. There are some C/C++ programs and shell scripts to running the tool. The documentation includes a simple guide about how to run the checker and some XML configuration files for checking different APIs.

VI. Experimental results

We wrote a custom test suite to evaluate the taint checker developed. We did focus on the following testing goals:

- The capacity for finding usage of tainted data in sinks.
- The number of false positives.
- Scalability.

In first case, the conservative implementation of checker makes that the capacity of finding usage of tainted data depends of the quality of description of the user configuration file. With well defined APIs of *sources*, *propagators*, *filters* and *sinks*, the tool is capable of detect any usage of tainted data in every test case with respect to the given configuration files.

The second test goal is more difficult to measure the number of false positives detected because in general it depends of the constraint solver capacity for prunning infeasible paths. The clang static analyzer constraint solver works on integers, booleans and memory locations (pointers and references), but it does't support (yet) floating point and other data types.

We find that in all tests, false positives result from infeasible paths, but not by the taint checker actions.

To measure scalability and the goals together, we applied the taint checker with a real software with a well known vulnerability which can be detected with taint analysis: The *OpenSSL hearthbleed*.

OpenSSL is a widelly used library for security that provides a lot of cryptographic functions, public key (certificates) management and platform independent files and network input-output API. OpenSSL versions from 1.0.1 throwgh 1.0.1f and the 1.0.2-beta contain a vulnerability that could allow a remote attacker to expose (read) sensitive data, possibly including user authentication credentials and secret keys, through incorrect memory handling in the `TLS` hearthbeat extension[23]. This bug could be found in some versions of Apache and nginx webservers and any other software which uses the OpenSSL library or utility commands.

The *Hearthbleed* vulnerability is an out of bounds memory read based on tainted data (data-length integers inside a TCP or UDP packets) being used as an argument of `memcpy`.

To find the OpenSSL *hearthbleed* vulnerability, we built a configuration file with the OpenSSL TCP/IP byte ordering swap macros (`ntos`, `ntohs` and `ntohl`) as sources of tainted data because operate on external inputs (network packets). The target is the `mempcy` function.

Below, we show the XML configuration file used for detecting the Hearthbleed vulnerability:

```
<?xml version="1.0" encoding="UTF-8"?>
  <TaintSources>
    <TaintSource>
      <method>ntohl</method>
      <params>
        <value>100</value>
      </params>
    </TaintSource>
   ...
  <TaintDestinations>
        <TaintDestination>
          <method>memcpy</method>
          <params>
            <value>2</value>
          </params>
        </TaintDestination>
  </TaintDestinations>
  <TaintFilters/>
</TaintChecker>
```

The results of our tainted checker were similar to result of *Coverity Scan* tool reported in [24]. In figure

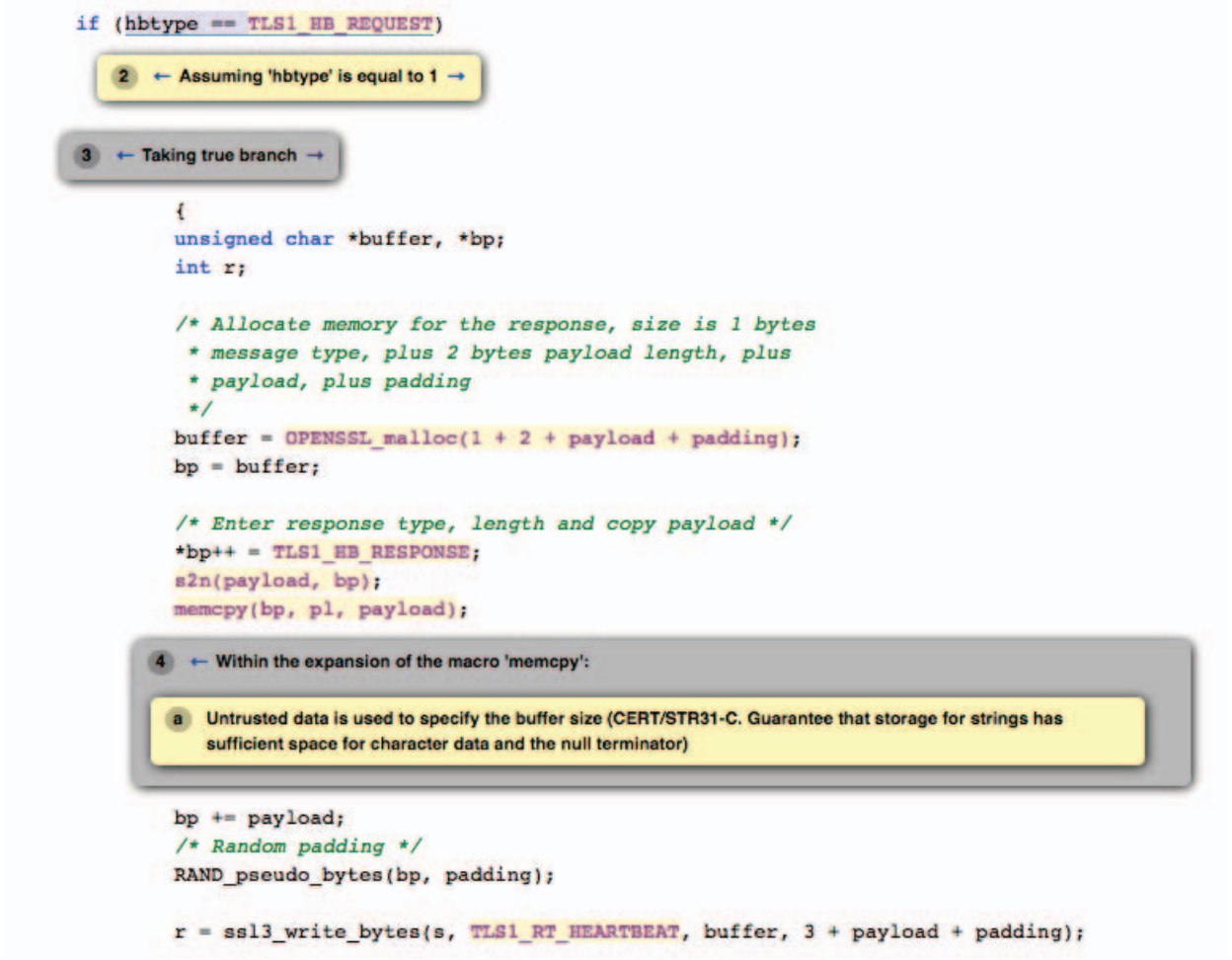

Fig. 7. Taint checker hearthbleed detection report.

VI, a fragment of a tainted data warning report is shown. This report shows a real case of tainted data (it is not a false positive). The times of analysis were less than compiling OpenSSL with full optimization (-O3) with no false positives for the heathbleed checking.

The falses positives reported in some cases is due to static analyzer core engine, mainly by the constraint solver limitations and to detect infeasible paths.

VII. Conclusions and Future Work

Taint checking been used effectivelly to find security vulnerabilies. The taint checker developed is capable of detect tainted data found in real software projects.

The taint checker developed was submitted to be integrated to the clang static analyzer (in alpha state) and it will included in the next release.

We are working in adding a mechanism to specify expressions (constraints) acting as filters to the configuration language, because in some cases a filter is not abtracted in a function but it is simple a condition in the source program.

The analyzer core engine provide the constraint set of each symbol (variable) for a given program point, so it should be possible to check if constraints contains the filter condition. For some known API as *memcpy()* or related functions, the tool could handle the constraints automatically, checking if storage for the first argument is greather than actual value of third argument.

This feature already was included as a replacement of the original taint checker provided in the last version of clang static analyzer. We are becoming the main mainterners of this checker and we are collecting

experimental results made from clang users working in big projects.

Reports generated by clang static analyzer contains precise information on the symbolic execution paths explored for each warning. We think this could be useful to test cases generation for checking if it was a false positive or not.

We are exploring the development of an external tool for this purpose by using SAT solving techinques or SMT solvers to concrete input values. Recently, there is a growing tendency to software correctness based on test cases generation based on symbolic execution as in [31] and [32].

REFERENCES

[1] *The GNU Compiler Collection (GCC)*. http://llvm.org/
[2] *LLVM*.http://llvm.org/
[3] *CLANG*. http://clang.org/
[4] OCLint. oclint.org/.
[5] S. Johnson. *Lint, a C program checker*. Computer Science Technical Report 65, Bell Laboratories, December 1977.
[6] D. Denning. *A Lattice Model of Secure Information Flow*. Fifth ACM Symposioum on Operating Systems Principles. November 1976.
[7] D. Denning and P. Denning. *Certification of Programs for Secure Information Flow*. Communications oof ACM, July 1977. Vol. 20, number 7.
[8] J. A. Goguen and J. Meseguer. *Security policies and security models*.Proc. IEEE Symp. on Security and Privacy. 1982.
[9] G. Smith. *Principles of Secure Information Flow Analysis*. Advances in Information Security. 27. Springer US. pp. 291-307.
[10] Patrick Cousot and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Porgrams by Construction or Approximation of Fixpoints*. Fourth ACM Symposium on Principles of Programming Languages. ACM, 1977, pp. 238-252.
[11] F. Schneider, G. Morriset and R. Harper. *A Language-Based Approach to Security*. R. Wilhelm (Ed.): Informatics: 10 Years Ahead. Springer Verlag. LNCS 2001. pp. 86-101.
[12] Robert. E. Strom and Shaula Yemini. *Typestate: A programming Language Concept for Enhancing Software Reliability*. IEEE Transctions on Software Engineering, Vol. SE 12, NO 1, January 1986.
[13] *Clang static analyzer*. http://clang-analyzer.llvm.org/
[14] *Coverity Scan Static Analysis*. https://scan.coverity.com.
[15] David Evans and David Larochelle. *Improving Security Using Extensible Lightweight Static Analysis.* IEEE Software magazine. Jan/Feb 2002.
[16] David Hovemeyer and William Pugh. *Finding Bugs is Easy*. ACM SIGPLAN Notices. Vol. 39, Issue 12. Pages 92-106. December 2004.
[17] James Newsome and Dawn Song. *Dynamic Taint Analysis for Automatic Detection, Analysis and Signature Generation of Exploits on Commodity Software*. 2004.
[18] James Newsome and Dawn Song. *Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software*. In Proceedings of the Network and Distributed Systems Security Symposium. 2005.
[19] *Open Wb Application Security Project*. https://www.owasp.org/
[20] T Reps, S Horwitz, and M Sagiv. *Precise interprocedural dataflow analysis via graph reachability*. Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Pages 49-61. ISBN:0-89791-692-1. 1995.
[21] Z Xu, T Kremenek, and J Zhang. *A memory model for static analysis of C programs*. Proceeding of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I Pages 535-548. Springer Verlag. 2010. ISBN: 3-642-16557-5 978-3-642-16557-3.
[22] Flemming Nielson and Hanne Riis Nielson. *Type and Effect Systems*. Correct System Design: Recent Insight and Advances. Lecture Notes in Computer Science, Springer Verlag. Pp 114-136. ISBN: 3-540-66624-9. 1999.
[23] *OpenSSL Heartbleed vulnerability (CVE-2014-0160)*. United States Computer Emergency Readiness Team. https://www.us-cert.gov/ncas/alerts/TA14-098A. 2014.
[24] Coverity blog. *On Detecting Hearthbleed with Static Analysis*. 2014.
[25] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto,J. Signoles, and B. Yakobowski. *Frama-c*. In International Conference on Software Engineering and Formal Methods (pp. 233-247). Springer Berlin Heidelberg.
[26] *Frama-C Software Analyzers*. http://frama-c.com
[27] A. Sabelfeld and A. Myers. *Language-Based Information-Flow Security*. IEEE Journal on Selected Areas in Communications. 2003.
[28] K. J. Biba. *Considerations for Secure Computer Systems*. Bedford, MA. USAF Electronic Systems Division. ESD-TR-76-372. 1977.
[29] Srijith Krishnan Nair, Patrick N. D. Simpson, Bruno Crispo and Andrew S. Tanenbaum. *A Virtual Machine Based Information Flow Control System for Policy Enforcement*. Electr. Notes Theor. Comput. Sci.. Vol. 197, number 1. 2008.
[30] Srijith K. Nair. *Remote Policy Enforcement Using Java Virtual Machine*. Phd Thesis. Vrije Universiteit. 2010.
[31] Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa and José Miguel Rojas. *Test Case Generation by Symbolic Execution: Basic Concepts, a CPL-Based Instance, and Actor-Based Concurrency*. SFM 2014, LNCS 8483, pp. 263-309, 2014.
[32] Tadahiro Uehara. *Exhaustive Test-case Generation using Symbolic Execution*. FUJITSU Sci. Tech. J., Vol. 52, No. 1, pp. 3440. January 2016.