Taint Dependency Sequences: a characterization of insecure execution paths based on input-sensitive cause sequences *

Dumitru Ceară and Laurent Mounier and Marie-Laure Potet
VERIMAG laboratory
University of Grenoble
38610 Gières, France
{*Dumitru.Ceara, Laurent.Mounier, Marie-Laure.Potet*}*@imag.fr*

***Abstract*—Numerous software vulnerabilities can be activated only with dedicated user inputs. Taint analysis is a security check which consists in looking for possible dependency chains between user inputs and vulnerable statements (like array accesses). Most of the existing static taint analysis tools produce some warnings on potentially vulnerable program locations. It is then up to the developer to analyze these results by scanning the possible execution paths that may lead to these locations with unsecured user inputs.**

We present a Taint Dependency Sequences Calculus, based on a fine-grain data and control taint analysis, that aims to help the developer in this task by providing some information on the set of paths that need to be analyzed. Following some ideas introduced in [1], [2], we also propose some metrics to characterize these paths in term of "dangerousness". This approach is illustrated with the help of the Verisec Suite [3] and by describing a prototype, called STAC.

***Keywords*-static taint analysis; vulnerability detection; test objectives.**

I. Introduction

Security has become a central issue for a large range of software systems, from the operating system kernel libraries to the top-level applications used on a regular basis by most computer users. Indeed, in spite of the progress achieved in terms of programming languages and techniques, it is still very difficult to completely avoid the presence of *software vulnerabilities* within large pieces of code. Moreover, most of these vulnerabilities are usually discovered *after* the software has been put on the market, which means that software editors have to deliver security patches on a regular basis.

To face this situation, it is necessary to provide tools that help the programmers to detect and possibly neutralize these vulnerabilities at several steps of the development process, either statically, during code reviews, or dynamically, during runtime validations or test campaigns.

A. Taint analysis

Numerous software vulnerabilities can be activated only with dedicated user inputs. For instance, a *buffer overflow* may occur within a file editor only if the input file contains special characters. Therefore, a well-established security analysis consists in looking for possible dependency chains between user inputs and vulnerable statements (like array accesses). Thus, when the value of a user dependent variable is used to perform a "critical" operation (from the security point of view), a potential vulnerability can be reported.

Such a dependency analysis, sometimes called *taint analysis*, can be performed either using static analysis techniques, or at runtime, by monitoring the current execution sequence. The former approach is more exhaustive, but may lead to *false positives*, whereas the latter needs to select first the execution sequences under check. Both techniques are implemented within several tools (see section VI).

B. Our contribution

Most of the existing static taint analysis tools produce some warnings on potentially vulnerable program locations. It is then up to the user to analyze these results by scanning the possible execution paths that may lead to this location with unsecured user inputs. Following some ideas proposed in [1], [2] our objective is to help the user in this task by providing some information on the set of paths that need to be analyzed. The main contributions of this paper can be summarized as follows:

- First, we propose a taint dependency calculus based on a formal *type system*, which associates a *taint environment* to each program location (namely a partition between tainted and untainted variables). This type system has been developed for a typical imperative programming language, and it encompasses most of classical constructs.
- Second, we extend this type system to produce a set of *Taint Dependency Sequences* (TDS for short) explaining why a variable is tainted: roughly speaking, a TDS is a set of program locations an execution path should traverse to reach a given location `l`, such that a given variable `v` is tainted at `l`.
- Third, we propose some *taint metrics* to characterize each TDS in terms of "dangerousness". For instance,

* **This work was partially developed in the context of the Vulcain Project (MSTIC 09-10, University of Grenoble).**

a TDS corresponding to a long dependency chain between a user input and a vulnerable statement, including both control and data dependencies, is more likely to be overlooked by the programmer.

Some of these contributions have been already implemented within a prototype tool and the results obtained are really encouraging with respect to mature existing taint-checkers [4].

The rest of the paper is organized as follows: in section II we illustrate in a more detailed manner our objective on a small (realistic) motivating example. In section III we describe our taint dependency calculus. In section IV we give some results obtained when applying this calculus to the motivating example, and in section V we discuss some implementation details of our prototype tools, and we give some experimental results. Section VI compares our approach with similar works.

II. Motivations and Contributions

A. The buildfname example

We chose C as the target language of our analysis because of its popularity for implementing efficient and largely scalable systems. Due to the fact that C is a very permissive language, a wide set of vulnerabilities can be found in the existing C code base. Among these, one of the most known vulnerabilities is the buffer overflow. The Verisec Suite [3] is a database created using known overflows which can be found in open source programs. It consists of a set of vulnerabilities discovered in real-life applications from which we may point out the Apache http server, Bind DNS server, the SAMBA suite, etc.

In the same suite we can find a few vulnerabilities regarding the **sendmail** general purpose email routing application. The function that we will address is the `buildfname` function, which stores the `login` data according to the user supplied `gecos` field (commonly referred to as *real-name*). The test case provided by Verisec abstracts the functionality of the function, preserving only the relevant parts for activating the vulnerability and also annotates the code with `/* BAD */` comments for the statements that may be exploited by an attacker. This simplified version appears in Listing 1.

```
1  void buildfname(char *gecos,
2   char *login,
3   char *buf)
4  {
5   char *p;
6   char *bp = buf;
7
8   for (p = gecos;
9          *p != ’\0’ && *p != ’,’
10         && *p != ’;’ && *p != ’%’;
11       p++) {
12
13     if (*p == ’&’) {
14       strcpy(bp, login);  /* BAD */
15       *bp = toupper(*bp);
16       while (*bp != ’\0’)
17         bp++;
18     } else {
19       bp++;
20       *bp = *p; /* BAD */
21     }
22   }
23   *bp = ’\0’; /* BAD */
24 }
```

Listing 1. buildfname simplified vulnerable function

The behaviour of this function is illustrated in Figure 1: the left-hand side gives the initial content of buffers `gecos` and `login`, whereas the right-hand side gives the final content of buffer `buf`. Pointer `p` is used to iterate on `gecos` and pointer `bp` to iterate on `buf`.

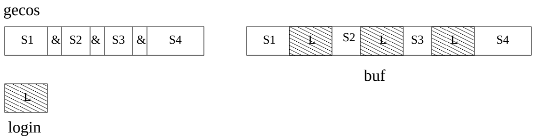

Figure 1. Behaviour of function `buildfname`

B. Buildfname Vulnerabilities

In our analysis we address mainly buffer overflow vulnerabilities of the following forms:

- writing outside the bounds of arrays using indices that may be influenced by the user (or, the equivalent, using pointers whose value can be changed by the user).
- using library functions that manipulate buffers (i.e., the `strcpy` function family when the source or destination buffers are controlled by the user).

There are three statements in the `buildfname` function that may be exploited by a malicious user. The first vulnerability (at line 14) can be activated either by using a value for `gecos` which contains a sufficient number of `’&’` characters in order to subsequently append the `login` argument to the fixed-size array `buf` or by supplying the `"&"` value for `gecos` and a large enough value for `login` to overflow `buf`.

The second vulnerability (line 20) can be activated in multiple ways. For instance, given an input value `S`, for `gecos` larger than the size of `buf`, the loop will always execute the `else` branch, thus leading to an overflow of the `buf` array. Another way to exploit this vulnerability is to execute the `then` branch a certain amount of times (without overflowing the `buf` array) and afterwards execute the `else` branch until `buf` is overflowed. This can be achieved by using an input of the form `"&&..&&S"`. Obviously these are not the only execution paths that activate this vulnerability. Different exploits can be built in order for the branches of the `if` statement to be executed alternately.

The vulnerability reported at line 23 represents (like the previous one) an out-of-bounds write bug which may be activated by the user by supplying a value for `gecos` which advances bp inside the loop to the end of the buf such that the last write will be done outside the bounds of the array.

C. Taint dependency analysis

As shown on the previous example, most buffer overflow vulnerabilities can be exploited by providing dedicated program inputs, allowing to activate specific execution paths. Detecting the existence of such potential exploits would be highly desirable, either during a review phase, or during a testing phase. To achieve that, we propose to perform a *taint dependency analysis* whose objective will be twofold:

1) to compute the variable taintness at each program location ;
2) to extract from the program source the whole dependency chains between each vulnerable statement and the corresponding inputs leading to this vulnerability.

This second information will be expressed by a so-called Taint Dependency Sequence (TDS, for short). Informally, a TDS can be viewed as a sequence of statements that must be executed in a specific order for a variable to become tainted at a given program location. Let us first illustrate more precisely these two notions on some examples.

```
1. x = get() ;
2. y = get() ;
3. w = y + 1 ;
4. z = x + w
5. t[z] = ...
```

Listing 2.

```
1. x = get() ;
2. y = get() ;
3. if (x >0 )
     4. y = 3 ;
   else {
     5. z = y ;
   }
6. t[z] = ...
```

Listing 3.

In listing 2 variables x, y, w and z are tainted at location 5. Moreover, the taintness of z comes from a *data dependency* between z and both x and w. x is tainted at location 1, whereas w is tainted because of a data dependency with y. Thus, the taintness of z at location 5 is expressed by the TDS set {<1,4>, <2,3,4>}.

In listing 3, variables x, y, and z are also tainted at location 6. Indeed, there are both a *control dependency* between z and x, since z is assigned within a conditional statement controlled by x, and a *data dependency* between z and y (at location 5). The taintness of z at location 6 is therefore explained by the TDS set {<1,3>, <2,5>}. In particular this set shows that tainting z can be obtained either by executing statements 1 and 3 (whatever the value of the condition), or by executing statements 2 and 5 (thus choosing a negative value for x).

```
1. x = 0 ;
2. y = 5 ;
3. while (x < 10) {
      4.  y = x+y ;
      5.  x = get() ;
}
6. t[y] = ...
```

Listing 4.

In listing 4, variables x and y are tainted at location 6. The taintness of y comes from several facts. First, there is a *data dependency* between y and x (which is directly tainted). Thus, after two iterations, y itself becomes tainted. Moreover, there is also a *control dependency* between y and x, since x becomes tainted after one iteration. The TDS set explaining why y is tainted after two iterations is {<5,4>, <5,3>}. For one iteration this set is empty. As a result, at least two iterations are required to taint y at location 6 and to activate a potential vulnerability. As we will see in Sect. III-C, a third iteration will be taken into account to compute the TDS, in order to capture the new dependency introduced at label 4 between the left-hand side and right-hand side occurrences of y.

III. Taint Dependency Sequence Calculus

We present here a calculus dedicated to Taint Dependency Sequences. For readability reasons, this calculus is composed of two parts: one part is dedicated to the taint calculus and the second one to TDS.

A. Notations

Let $Taint$ be the set $\{T, U\}$ where T denotes tainted values (depending on the user-input) and U denotes untainted values. $Taint$ is ordered in the following way: $T < U$. The operator $\otimes$ between taint values is defined by:

$$T \otimes\ x = T \qquad U \otimes\ x = x \qquad x \otimes\ y = y \otimes x$$

A taint environment is a function $\Gamma : Name \rightarrow Taint$ that associates a taint value to each program variable name. The operator $\otimes$ can be extended to taint environments in the following way:

$$\Gamma = \Gamma_1 \otimes\ \Gamma_2 \Leftrightarrow \Gamma = \lambda x\ .\ \Gamma_1(x) \otimes \Gamma_2(x)$$

In the rest of this section we use the notation $x \mapsto e$ to denote the association of e to the element x and $f[g]$ to denote the function f overridden by the function g: $f[g](x) = g(x)$ if $x \in dom(g)$ and $f[g](x) = f(x)$ otherwise.

B. Taint and Taint Dependency Sequences Judgements

Our taint calculus is based on the following judgements (with c a command and e an expression):

$$\Gamma, \alpha \vdash c : \Gamma'$$
$$\Gamma \vdash e : \tau$$

Γ represents the taint environment before the execution of c (or the evaluation of e) and α the current control flow (T if the command is under the scope of a tainted condition, U otherwise). Γ' is the taint environment obtained after c and τ the taintness of e.

Taint analysis is very close to non-interference [5], [6] so the correctness property states that untainted values remain unchanged when input values change (a similar property can be stated for expressions: if e is evaluated as untainted from Γ then its value is not sensitive to input data variation.) In particular, if all variables are tainted the correctness property obviously holds. Due to the correctness definition the following subtyping rule can be stated:

$$\frac{\Gamma, \alpha \vdash c\colon \Gamma' \qquad \Gamma'' \leq \Gamma'}{\Gamma, \alpha \vdash c\colon \Gamma''}$$

with $\Gamma'' \leq \Gamma'$ iff $\Gamma''(x) \leq \Gamma'(x)$ for each x in $Name$.

According to the taint calculus we will compute taint dependency sequences describing, for each variable and each program point, a set of sequences of assignments or conditions that can explain why the variable is tainted at this point.

A TDS is of the form $< l_1, \ldots, l_n >$ with l_i a statement label. A TDS environment is a function Λ that associates a set of taint dependency sequences to each program variable name. Let t be a TDS at label l and P the concerned program: t denotes the subset of paths of P starting from its entry point to the label l and containing all labels of t in the same order. Intuitively $\Lambda(x)$ at label l denotes all paths of P effectively tainting x at label l. *A contrario*, a path p that does not correspond to a TDS in $\Lambda(x)$ does not taint x in any way: the value of x obtained with p at label l is insensitive to input variations.

Our TDS calculus is based on the two following judgements (with c a command and e an expression):

$$\begin{array}{l} \Lambda, \phi \vdash c : \Lambda' \\ \Lambda \vdash e : \lambda \end{array}$$

Λ represents the set of TDS associated to each variable before the execution of c (or the evaluation of e) and ϕ represents the set of TDS justifying the taintness of conditions that may control the taintness of c, if any.

Finally our taint and Taint dependency Sequences judgements will be linked by the following global invariants (where $Ass(c)$ denotes the set of variables that are assigned in command c):

$$\begin{array}{ll} Inv_1 : & x \in Ass(c) \Rightarrow \ \Gamma'(x) \leq \alpha \\ Inv_2 : & \phi = \emptyset \Leftrightarrow \ \alpha = U \\ Inv_3 : & \Gamma(x) = U \Leftrightarrow \ \Lambda(x) = \emptyset \\ Inv_4 : & \tau = U \Leftrightarrow \ \lambda = \emptyset \end{array}$$

C. Taint and TDS Calculus

We present here a calculus for a small language (see Sec. V for our real implementation). The considered language is:

lvalue → idf | *lvalue
exp → num | lvalue | op(list_exp) | get()
comm → (label) lvalue := exp | comm ; comm
comm → (label) if exp then comm else comm
comm → (label) while exp do comm

Labels associated to control flow commands will be attached to the condition. Expressions of the form op(list_exp) denote any operator such as unary or binary arithmetic operators or the unary * operator. Expression get() is a particular function that reads values.

1) Literal: each literal value (string literal or numeric literal) is considered untainted.

$$\Gamma \vdash n : U \qquad\qquad \Lambda \vdash n : \emptyset$$

2) Variable: The taint value and the explanation of this taint are directly extracted from the current environments Γ and Λ.

$$\Gamma \vdash x : \Gamma(x) \qquad\qquad \Lambda \vdash x\colon \Lambda(x)$$

3) Operator: The taint type and the TDS set of an expression is obtained from the taint types and TDSs of each subexpression.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash op(e_1, \ldots, e_n) : \bigotimes_{i \in 1..n} \tau_i}$$

$$\frac{\Lambda \vdash e_1 : \lambda_1 \qquad \ldots \qquad \Lambda \vdash e_n : \tau_n, \lambda_n}{\Lambda \vdash op(e_1, \ldots, e_n) : \bigcup_{i \in 1..n} \lambda_i}$$

4) Get expression: obviously the returned value is input-sensitive.

$$\Gamma \vdash get() : T \qquad\qquad \Lambda \vdash get() : \{<>\}$$

5) Simple assignment: When an assignment is executed the mapping for the assigned variable in Γ is changed according to the taint type of the right hand side expression. Furthermore, if this assignment is under the scope of a tainted condition ($\alpha = T$) then x becomes tainted, independently of e.

$$\frac{\Gamma \vdash e\colon \tau}{\Gamma, \alpha \vdash (l) \ x := e\colon \Gamma_{[x \mapsto \tau \ \otimes \ \alpha]}}$$

For instance the assignment `y=5` of listing 4 (label 2) does not taint `y` whereas the assignment `y=3` of listing 3 (label 4) leaves `y` tainted because we are under the scope of the tainted condition `x>0`.

If the assignment taints the left hand side variable, then the TDS set required to taint the right hand side expression and the current label are associated to the assigned variable. If $\alpha = T$ then ϕ also explains the taintness of x, as explained above.

$$\frac{\Lambda \vdash e\colon \lambda}{\Lambda, \phi \vdash (l) \ x := e\colon \Lambda_{[x \mapsto app(l, \lambda) \cup \phi]}}$$

The function app is defined by $app(l, \lambda) = \{< p, l >| \ p \in \lambda\}$, adding the label l at the end of each sequence in λ. In particular $app(l, \emptyset) = \emptyset$.

6) Assignment with dereferencing: The following rules treat assignments of the form $*p := e$, with p a variable name. If p is initially tainted it remains tainted. The two following rules can be easily extended to any chain of dereferencing.

$$\frac{\Gamma \vdash e \colon \tau}{\Gamma, \alpha \vdash (l) \;\; *p := e \colon \Gamma_{[p \mapsto \Gamma(p) \otimes \tau \otimes \alpha]}}$$

$$\frac{\Lambda \vdash e \colon \lambda}{\Lambda, \phi \vdash (l) \;\; *p := e \colon \Lambda_{[p \mapsto \Lambda(p) \cup \; app(l,\lambda) \cup \; \phi]}}$$

In case of a structured object (as pointer or array structure) any modification of some parts of this object affects the object in the whole. Since it is not possible to statically determine if a new modification concerns a part already updated or not, the current taintness of this object must be taken into account. A more precise algorithm should require a fine-grain memory model, as in [7] for instance.

7) Sequencing: When executing a sequence of commands the two environments Γ and Λ are modified according to the two commands in the sequence.

$$\frac{\Gamma, \alpha \vdash c_1 \colon \Gamma_1 \qquad \Gamma_1, \alpha \vdash c_2 \colon \Gamma_2}{\Gamma, \alpha \vdash c_1; c_2 \colon \Gamma_2}$$

$$\frac{\Lambda, \phi \vdash c_1 \colon \Lambda_1 \qquad \Lambda_1, \phi \vdash c_2 \colon \Lambda_2}{\Lambda, \phi \vdash c_1; c_2 \colon \Lambda_2}$$

8) Control flow commands: for each if statement the type environment obtained after applying the inference rule will have to keep trace of all the assignments made on both branches ($\Gamma_1 \otimes \Gamma_2$). Moreover, as soon as the condition is tainted each variable assigned in one branch of the condition becomes tainted (Γ') and the control flow dependence scope is extended by e ($\alpha \otimes \tau$ for the taintness calculus and $\phi \cup app(l, \lambda)$ for the TDS calculus).

$$\frac{\begin{array}{c}\Gamma \vdash e \colon \tau \\ \Gamma' = \Gamma[\{x \mapsto \tau \otimes \Gamma(x) \mid x \in Ass(c_1) \cup Ass(c_2)\}] \\ \Gamma', \alpha \otimes \tau \vdash c_1 \colon \Gamma_1 \\ \Gamma', \alpha \otimes \tau \vdash c_2 \colon \Gamma_2\end{array}}{\Gamma, \alpha \vdash (l) \; if \;\; e \;\; then \;\; c_1 \;\; else \;\; c_2 \colon \Gamma_1 \otimes \Gamma_2}$$

As previously, $Ass(c)$ denotes the set of variables that are modified in the command c. For instance in Listing 3, variables `y` and `z` are tainted inside the two branches because the user can control their values by controlling the value of `x`. Then the TDS <1, 3> that explains why the condition is tainted also explains the taintness of `y` and `z`.

$$\frac{\begin{array}{c}\Lambda \vdash e \colon \lambda \\ \Lambda' = \Lambda[\{x \mapsto app(l, \lambda) \cup \Lambda(x) \mid x \in Ass(c_1) \cup Ass(c_2)\}] \\ \Lambda', \phi \cup \; app(l, \lambda) \vdash c_1 \colon \Lambda_1 \\ \Lambda', \phi \cup \; app(l, \lambda) \vdash c_2 \colon \Lambda_2 \\ \Lambda'' = \{x \mapsto \Lambda_1(x) \cup \Lambda_2(x) \mid x \in Name\}\end{array}}{\Lambda, \phi \vdash (l) \; if \;\; e \;\; then \;\; c_1 \;\; else \;\; c_2 \colon \Lambda''}$$

9) Loop command: For the loop statement the resulting taint environment can be characterized by a greatest fixpoint:

$$\frac{\begin{array}{c}\Gamma \vdash e \colon \tau \\ \Gamma, \alpha \vdash c \colon \Gamma \\ \forall x \in Ass(c) \; . \; \Gamma(x) \leq \tau \otimes \alpha\end{array}}{\Gamma, \alpha \vdash (l) \; while \;\; e \;\; do \;\; c \colon \Gamma}$$

This greatest fixpoint will be computed by iteration using the subtyping rules (Section III-B) and the following judgements:

$$\begin{array}{c}\Gamma_0, \alpha \vdash (l) \; while \;\; e \;\; do \;\; c \colon \Gamma' \\ \Lambda_0, \phi \vdash (l) \; while \;\; e \;\; do \;\; c \colon \Lambda'\end{array}$$

Variables assigned within a loop statement are tainted either because of control dependencies with the loop condition (as soon as this condition becomes tainted itself), or because of data dependencies with other variables (inside or outside the loop). Dependency chains can be much longer in the second situation. In order to catch these longest chains we first compute the minimal number of loop iterations required to detect all data dependencies. This is obtained by rewriting *while* statements as blocks of nested *if* statements according to the following definition:

$$while\_if(e, c, n) = \begin{cases} \begin{array}{l} if \;\, e \; then \\ \quad c; while\_if(e, c, n-1) \\ else \; skip \end{array} & n > 0 \\ skip & n = 0 \end{cases}$$

Thus, with $\Gamma, \alpha \vdash skip : \Gamma$ and $\Lambda, \phi \vdash skip : \Lambda$, the algorithm we propose is:

1) Compute $\Gamma_i, \alpha \vdash$ c $: \Gamma_{i+1}$ starting with with $i = 0$. Let $n = i + 1$, with i the smallest value such that $\Gamma_{i+1} = \Gamma_i$, be the number of iterations required to establish the fixpoint.
2) Compute $\Lambda_0, \phi \vdash while\_if(e, c, n) : \Lambda'$.
3) Compute the final taint environment
$\Gamma'(x) = \begin{cases} T \text{ if } \Lambda'(x) \neq \emptyset \\ U \text{ if } \Lambda'(x) = \emptyset \end{cases}$ for each x in $Name$.

Let us consider the example on listing 4. At step 1, three iterations are required in order to stabilize the taintness calculus (the first iteration taints `x`, the second one taints `y` and the third one establishes the fixpoint). At label 6, the set $\Lambda(y)$ contains the following TDS sets: $\emptyset$ after the first iteration, {`<5,4>`, `<5,3>`} after the second iteration extended by {`<5,3,4>`, `<5,3,3>`, `<5,4,4>`} after the third one.

D. *Data-input sensitive vulnerability detection*

Based on our type system, we developed an algorithm that computes the taint value of each variable at each label with their associated TDS sets. These informations are exploited to detect potential vulnerable statements. Let us consider some examples:

- a statement of the form $t[i] = e$ (or $*t = e$) may provoke a sensitive user input buffer overflow if, in the current Γ environment, the following hold: $\Gamma \vdash i : T$ or $\Gamma \vdash t : T$.
- a call of the form $strcpy(dest, source)$ or $strcat(dest, source)$ is potentially vulnerable if $\Gamma \vdash dest : T$ or $\Gamma \vdash source : T$.

IV. From TDS to validation objectives

A. *TDS metrics*

An immediate application of our static taint dependency analysis is to provide some useful guides on the execution paths that could be used at runtime for finding potential exploits. In particular the TDS set we compute at each vulnerable location could be viewed as *validation objectives,* whose purpose is to select the execution paths to be exercised during a code review or a test campaign.

First, as said in section III-B, each execution path leading to a vulnerable location (l) which is not "covered" by a TDS can be considered as *safe*. Indeed, the values produced by this execution path are not sensitive to any user inputs[1]. Thus, the TDS we compute give us (an over-approximation of) the whole set of non-safe execution paths to be considered. This information provides useful indications by delineating the path space to be considered.

Second, the TDSs associated to each potential vulnerability also help us to classify the non safe execution paths according to several criteria. For instance, they help to characterize how "dangerous" a given execution path is, i.e., what are the chances for this path to correspond to an actual exploit. Such a measure can be obtained by associating a set of *metrics* to each TDS. Some possible metrics could be:

The TDS size: the longer a TDS is, the more precise are the associated execution paths, and the longer the taint dependency chains they involve are. We could assume that long taint dependency chains are more likely to correspond to exploits, since they have more chances to not be fully controlled by the programmer with appropriate sanity checks (as pointed out in [2]).

The taintness source: to each TDS can be associated a set of variables corresponding to *taint sources*. These variables are the ones that are directly tainted by means of input functions. We can assume that some input streams are more likely to carry unsecured data, or are easier to access from non trusted users. Corresponding execution paths are therefore more critical.

The structure of the dependency chain: variable taintness may have several causes: data dependencies with taint variables, control dependencies with tainted expressions, or combinations of both. Here again, we can assume that complex dependency chains (i.e., with numerous combinations between control and data dependencies), involving a large amount of data, are more likely to escape from the programmer attention, and therefore may lead to potential exploits.

The number of loop iterations: many buffer overflow exploits are obtained by increasing array indexes beyond their expected values. This is usually achieved by iterating on loop bodies. Therefore, TDS covering loop bodies are more likely to correspond to potential exploits. Note also that, when a taint dependency chain depends on loop iterations, the TDS set we produce gives the *minimal* number of iterations required to taint all the elements of the chain.

All these metrics can be easily obtained from basic information associated to each TDS. In the following section we give some examples of TDS obtained from the `buildfname` function, the corresponding metrics, and their relationship with the exploits we identified in section II.

B. *Back to the* `buildfname` *example*

The control flow graph of the `buildfname` function (listing 1) is depicted in Figure 2. For a better understanding we have associated to each statement the line number as its label. Also, the `for` loop has been changed into a `while` loop (for uniformity) by adding a block for the initialization step (label 8) and a block for the increment step (label 11). The two initial blocks (labels 1 and 2) are used to explicitly show the taintness of the two arguments. The `strcpy(bp, login)`, from the taintness point of view, behaves as the assignment `*bp=*login`. Finally, we consider in the following that the function `toupper` (statement 15) is transparent from the taintness point of view (the taint of its result is the one of its argument).

As stated in section II-B, three potential vulnerabilities are identified in this example. All theses vulnerabilities are detected by our type checking algorithm. Two iterations on loop `9` and one iteration on loop `16` are required to compute the fix-point of function Γ. We focus on vulnerability 14.

C. *TDS metrics for vulnerability 14*

The statement `strcpy(bp, login)` is marked as "vulnerable" by our analysis for two distinct reasons: both variables `login` and `bp` are tainted when executing this statement.

Variable `login` is a user input (at location `2`), and it is never assigned elsewhere in the function. Therefore $\Lambda($`login`$) = \{$`<2>`$\}$. Regarding $\Lambda($`bp`$)$, the situation is more complex since there are numerous dependency chains

[1] of course, some vulnerabilities can also be activated independently to any user input, but they are not in the scope of this paper.

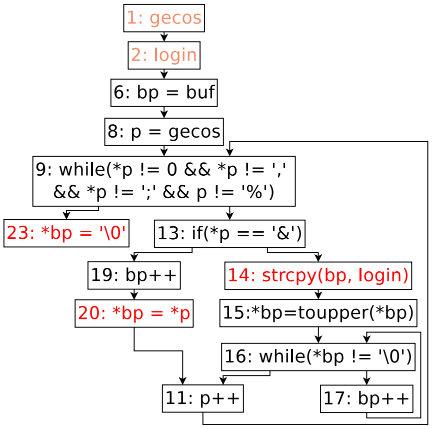

Figure 2. buildfname CFG

source	data	control only with 9(13)	control with 16	#TDS
login	2	0	2	4
gecos	1	18	6	25
#TDS	3	18	8	29

Table I
TDS METRICS FOR $\Lambda(bp)$ AT LABEL 14

explaining why variable bp is tainted at location 14. Indeed, our algorithm produces 29 (21 for the first iteration, 8 until label 14 is reached again during the second iteration) distinct TDS. Table I supplies a classification of this TDS set based on the source (starting with 1 for a dependency with gecos and starting with 2 for a dependency with login) and on the type of dependency (pure data dependency vs control and data dependencies).

D. Application to exploit generation

First, some useful indications are easy to extract from Table I, either to guide a code review or to prepare a test campaign dedicated to the search of actual exploits. Such indications could help to better manage the time budget by focussing first on the most critical parts, for instance:

- The gecos value can be considered as easier to choose/modify by an untrusted user than the login value (usually set by the administrator). Thus, execution paths corresponding to TDS tainted by gecos could be privileged.
- Among them, several TDS have rather long dependency chains (of size 5 or 6), including combinations of data and control dependencies. The corresponding paths are longer, their control flow is more complex, and they are therefore more likely to lead to exploits.

However, a deeper analysis of the metrics summarized in Table I allows to classify the TDS into four distinct groups, and to retrieve the potential exploit we identified in section II-B.

A first TDS set is obtained by considering the (pure) data dependencies between bp and login (first line and first column): <2,14>, <2,14,17>. The corresponding exploit consists in appending a sufficiently large login value (sufficiently often) to overflow buf. Note that for tainting bp at least one complete iteration of loop 9 must be performed.

A second TDS set is obtained by considering the (pure) data dependencies between bp and gecos (second line, second column). This set actually contains only one TDS, <1,8,20>, because bp can only be tainted by gecos, in the first iteration, with the direct assignment at label 20. The corresponding exploit consists in overflowing buf by supplying a sufficiently large gecos value. Note that the gecos input must contain at least one '&' character in order for bp to become tainted.

A third TDS set is obtained by considering that the taintness of bp can come from a control dependency with p at location 9 and 13. Column 3 does not contain any TDS starting with 2, meaning that there is no data flow from login to p. Exploiting the control dependency between bp and p necessarily requires supplying an appropriate gecos value. TDS examples are: <1,8,9>, <1,8,11,9,13>, <1,8,9,13,17>, etc. The corresponding exploit consists in choosing a gecos value leading to a sufficient number of iterations of loop 9 to overflow buf.

A fourth TDS set is obtained by considering that the taintness of bp can come from a control dependency with bp at location 16 (column 4). The corresponding exploit would consist in providing a sufficiently large login value to execute sufficiently often statement 17 to overflow buf. Note that executing statement 14 has exactly the same effect (loop 16 does not bring any new exploit), but a more semantic analysis would be required to detect that.

Each TDS corresponds to a set of possible execution paths on which the variable can become tainted. The whole TDS set could be supplied to the developer in order to describe all the possible executions that must be analyzed. However, a classification of the computed TDS can be automatically created (using the metrics described earlier), allowing the developer to decide which are the TDS classes on which further analyses should focus.

V. STAC IMPLEMENTATION

We present here a prototype, STAC[2], which implements the TDS calculus presented earlier, extended to cover a larger subset of the C language (handling aliasing and the existence of procedures).

A. Frama-C

STAC uses the Frama-C platform [7] as its front-end, an extensible platform for source-code analysis of C programs,

[2] http://code.google.com/p/tanalysis/

built on top of CIL [8]. It can also be seen as a collaborative framework for developing static analyzers. The collaborative approach allows analyzers to use the results already computed by other analyzers in the framework. The Frama-C engine supports all the constructs of the C language. However, each analyzer is allowed to define the subset of the language it will address.

B. Addressing the C language

Due to the fact that our analysis aims at being sound and because we target C programs, the presence of pointers requires additional alias information. We use the implementation provided in [9] for Steensgaard's algorithm for almost linear inter-procedural points-to analysis ([10]).

The small language we provided in Sec. III does not cover all the available constructs of the C language such as arrays, structures and unions. In STAC we consider these kinds of constructs as objects so, whenever we assign a taint value to an entry of an array (or to a structure or union field) we either change the value of the whole object to T (if the right-hand side was tainted) or we keep the previous taint value of the array in a manner similar as described for the assignment with dereferencing in Sec. III. All these extensions are over-approximations of our analysis. However, none of them invalidates the correctness properties we have stated before.

C. Intra and Inter procedural analysis

For each procedure in the analyzed program, STAC performs a flow sensitive analysis on the CFG[3] high-level representation of the procedure (as provided by Frama-C). The analysis applies the type system rules to all the statements in the CFG starting with the entry point of the procedure. In this way, when the analysis is performed for a given procedure, we compute a taintness environment for its exit point (the effect of the execution according to taintness). We will refer to this environment as the *summary* of the procedure. The taintness values in the summary will be dependent on the taintness of the formal parameters of the procedure.

Regarding the inter-procedural aspect of our calculus, STAC performs a context sensitive analysis. When a procedure call-site is reached the effect of the summary is applied to the environment associated to the call-site environment. The order in which STAC analyzes the procedures is given by the reverse topological ordering of the strongly connected components in the call-graph provided by Frama-C (in a manner similar to the one presented in [4]). Computing the summaries for each procedure inside a strongly connected component is equivalent to a fix-point computation (as each component corresponds to a loop in the call-graph of the program).

D. Experiments

In order to evaluate the performance of our implementation we have performed experiments using the NIST Samate reference dataset ([11]), but also some widely used Linux applications like **sendmail** and **mailx**. We used the set of vulnerability patterns given in Sec. III-D.

The Samate dataset contains small vulnerability examples and after running our implementation on 300 test-cases we obtained an average of 46.5% tainted statements. In order to exploit the taintness information for detecting potential vulnerable statements we use the set of patterns given in Sec. III-D. The results we obtained can be seen in table II. The **#loc** column indicates the number of lines of code in the analyzed programs whereas **% tainted stmts** and **% vulnerable stmts** indicate the percentage of tainted statements computed by STAC and respectively the percentage of potential vulnerable statements according to the previously described patterns.

prog	#loc	% tainted stmts	% vulnerable stmts
sendmail	83499	67.2	7.9
mailx	10171	61.6	10.5

Table II
STAC RESULTS FOR SENDMAIL AND MAILX.

Our results are similar to the ones obtained in [4] using the context sensitive approach to compute user-input dependencies. The small differences can be explained by the fact that results are influenced by the way the library functions are annotated; if a summary for a library call is not provided, STAC assumes the worst case and taints the return values of the function.

VI. Related works

Computing user input dependencies, or variable taintness[4], to detect potential vulnerabilities within a software is not an original idea. Thus, a large amount of work has been already published on this topic. We briefly review in this section the most commonly used approaches focusing more on the works that are close to the one we proposed in this paper.

A. Dynamic taint-analysis

The notion of *taint variable* has been introduced within the PERL language, with the use of a special execution mode called "taint mode". Within this mode, the PERL interpreter propagates information about tainted data (i.e., coming from untrusted sources) across program assignments and raises a security error when an insecure system call occurs. This idea of computing variable taintness *at runtime* has been generalized in several tools like [12], [13], [14], see [15] for a more complete survey.

[3] Control Flow Graph

[4] the difference between these two notions is that in taint-analysis one usually assumes a notion of sanitization.

Dynamic taint-analysis provides several advantages since it considers a concrete execution path, with all information available regarding the current variable valuations. Thus, sanity checks can be handled accurately, avoiding many false positives. However, since each analysis is reduced to a single (current) execution path, its coverage level may remain very weak and control dependencies cannot be fully taken into account. Therefore this technique may lead to numerous false negatives (i.e., unrevealed vulnerabilities).

Because we aim to be complete in terms of vulnerability detection, a dynamic taint analysis is not appropriate for us. Nevertheless, since we want to build some precise guides for the generation of exploits, we have to be as precise as possible, and close to the notion of execution paths. Thus, our TDS calculus is similar to a dynamic taint analysis in the sense that variable taintness is attached to a subset of execution paths.

B. *Static taint-analysis*

Another approach to compute variable taintness is to use static analysis techniques, allowing to take into account the whole set of a program execution sequences.

Static taint analysis can be based on several formalizations: *program dependence graphs* [16] and [17], program slicing techniques [18], or type systems [19] as used in the CQual tool [20]. For scalability reasons, internal representations as SSA (Static Single Assignment), GSA (Gated Single Assignment) or aSSA (Augmented Static single Assignment) are often used [4]. Note also that static taint analysis techniques are now also widely investigated for specific vulnerability detections within web based applications [21], [22], [23].

Most of static taint analysis tools are only dedicated to user input data dependencies [24]. Since we strongly believe that this approach is not sufficient to detect every potential exploit our taintness calculus fully implements user input control dependencies. As pointed out in Sec. V, our work is quite close to the PARFAIT tool [4]: we handle (sensitive) inter-procedural analysis in a similar way, we use a (simple) may-alias analysis to deal with pointers, and the behaviours of C library functions are abstracted by function summaries. However, their taintness calculus for control dependency does not exactly correspond to the one used in information flow analysis ([25], [16]) and its clear underlying notion of correctness we aim to encompass (see [4] for a finer comparison).

C. *Computing path information*

As discussed in this paper, from a security checking point of view it is desirable to extend results on variable taintness with information on the corresponding execution paths. This idea has been already developed in a few works:

- In [2], lengths of user dependency chains between a potential vulnerability and the corresponding user input are computed. This measure provides metrics allowing us to classify execution paths leading to a vulnerability (the longest being the most dangerous). However, loop executions are not precisely taken into account in this computation.
- Execution paths are also considered in [1], from a qualitative point of view. In particular they propose a path classification into 4 categories: infeasible, safe (sanitized), vulnerable (tainted), user-independent and “don’t-know”. Such a classification is obtained using a backward analysis technique in conjunction with a solver to compute over-approximations of variable valuations (to detect infeasible or safe paths).

The work we presented here borrows some ideas from [2] and [1]. In particular we tried to provide a finer classification of the set of vulnerable paths than the one proposed in [2] by taking into account the possible combinations of user input sources to taint a given expression. Furthermore, we also proposed some metrics to characterize these paths from a quantitative point of view.

VII. CONCLUSION

We have proposed a Taint Dependency Sequence Calculus whose objective is twofold: it relies on a fine-grain taint variable analysis (including data and control dependencies), and it produces some explicit representation of the sets of dependency chains from each input statements to each potentially vulnerable program locations.

We believe that such a calculus provides a useful “basic block” within a more complete vulnerability detection environment. First, dealing with combinations of both control and data dependencies increases the chance of detecting unforeseen program behaviours, since they are hard to master by developers. In particular they offer more subtle ways to build exploits by controlling which assignments will take place or not (e.g., the number of & in gecos in our example). Moreover, by making such combinations explicit in terms of execution paths, we give a concrete representation of the parts of code that need to be further analyzed (through a code review, or a test campaign). An interesting feature is that this calculus also provides some precise indications regarding the *minimal* number of loop iterations to be performed in order to taint a loop body. This should improve the classical test generation process, where loop iterations are performed randomly.

This work can be continued in several directions. First, it would be interesting to study how the TDS (and TDS metrics) we produce could be turned into concrete *test objectives*, to be used as input within a test process. This step could for instance take into account some other kinds of analysis (such as symbolic evaluation for instance), in particular to deal with sanity checks (and remove some false positives). On a similar basis, the TDS we produce could also be used to guide some dynamic program execution, possibly by combinations between concrete and symbolic data representations (like in *concolic execution*, [26]). To do so, it would be useful to define first some clear notion of equivalence (or subsumption) between TDS.

Finally, as described in Sec. V we provide a prototype implementation of our calculus. In order to improve performance, we could either perform the calculus in a ”lazy” manner such that environment computations are evaluated only when needed or use symbolic efficient data structures for handling the environments and TDS sets.

References

[1] W. Le and M. L. Soffa, “Refining buffer overflow detection via demand-driven path-sensitive analysis,” in *PASTE ’07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2007, pp. 63–68.

[2] C. Nagy and S. Mancoridis, “Static security analysis based on input-related software faults,” in *Conference on Software Maintenance and Reengineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2009.

[3] http://se.cs.toronto.edu/index.php/Verisec_Suite.

[4] B. Scholz, C. Zhang, and C. Cifuentes, “User-input dependence analysis via graph reachability,” in *Source Code Analysis and Manipulation, IEEE International Workshop on*, Los Alamitos, CA, USA, 2008, pp. 25–34.

[5] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *J. Comput. Secur.*, vol. 4, no. 2-3, pp. 167–187, 1996.

[6] J. A. Goguen and J. Meseguer, “Security policies and security models,” in *IEEE Symposium on Security and Privacy*, 1982.

[7] http://frama c.cea.fr.

[8] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “Cil: Intermediate language and tools for analysis and transformation of c programs,” in *CC ’02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 213–228.

[9] http://hal.cs.berkeley.edu/cil/.

[10] B. Steensgaard, “Points-to analysis in almost linear time,” in *POPL ’96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 32–41.

[11] http://samate.nist.gov/SRD.

[12] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” in *Proceedings of the Network and Distributed System Security Symposium, San Diego, California*. The Internet Society, 2005.

[13] W. Xu, S. Bhatkar, and R. Sekar, “Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks,” in *USENIX-SS’06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006.

[14] J. Clause, W. Li, and A. Orso, “Dytan: a generic dynamic taint analysis framework,” in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2007, pp. 196–206.

[15] W. Chang, B. Streiff, and C. Lin, “Efficient and extensible security enforcement using dynamic data flow analysis,” in *Proceedings of the 15th ACM conference on Computer and Communications Security*. New York, NY, USA: ACM, 2008, pp. 39–50.

[16] C. Hammer, J. Krinke, and G. Snelting, “Information Flow Control for Java Based on Path Conditions in Dependence Graphs,” in *In IEEE International Symposium on Secure Software Engineering*, 2006.

[17] G. Snelting, T. Robschink, and J. Krinke, “Efficient path conditions in dependence graphs for software safety analysis,” *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 4, 2006.

[18] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar, “Interprocedural analysis for privileged code placement and tainted variable detection,” in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3586. Springer, 2005, pp. 362–386.

[19] J. S. Foster, T. Terauchi, and A. Aiken, “Flow-sensitive type qualifiers,” in *PLDI ’02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM, 2002, pp. 1–12.

[20] http://www.cs.umd.edu/ jfoster/cqual/.

[21] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities (short paper),” in *IN IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, 2006, pp. 258–263.

[22] G. Wassermann and Z. Su, “Sound and precise analysis of web applications for injection vulnerabilities,” in *PLDI ’07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2007, pp. 32–41.

[23] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, “Taj: effective taint analysis of web applications,” in *PLDI ’09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009, pp. 87–97.

[24] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov, “Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities,” in *CSF ’09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 186–199.

[25] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003.

[26] P. Godefroid, N. Klarlund, and K. Sen, “Dart: directed automated random testing,” in *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 213–223.