LiSTT: An Investigation into Unsound-incomplete Yet Practical Result Yielding Static Taintflow Analysis

Sanjay Rawat
International Institute of Information Technology
Hyderabad, INDIA
Email: sanjay.rawat@iiit.ac.in

Laurent Mounier*
Verimag
University of Grenoble, France
Email: Laurent.Mounier@imag.fr

Marie-Laure Potet*
Verimag
University of Grenoble, France
Email: Marie-Laure.Potet@imag.fr

***Abstract*—Vulnerability analysis is an important component of software assurance practices. One of its most challenging issues is to find software flaws that could be *exploited* by malicious users. A necessary condition is the existence of some *tainted* information flow between tainted input sources and vulnerable functions. Finding the existence of such a taintflow *dynamically* is an expensive and nondeterministic process. On the other hand, though *static analysis* may explore (theoretically) all the tainted paths, scalability is an issue, especially in the view of *complete- and sound-ness*. In this paper, we explore the possibilities of making static analysis scalable, by compromising its complete- and sound-ness properties and yet making it effective in detecting taintflows that lead to vulnerability exploitation. This technique is based on a combination of call graph slicing and data-flow analysis. A prototype tool has been developed, and we give experimental results showing that this approach is effective on large applications.**

***Index Terms*—Security testing (assurance), vulnerability analysis, program chopping, static taint analysis, binary code.**

I. Introduction

Software assurance is the process of testing (verifying) the system under test (*aka* SUT) to establish (with reasonably high confidence) that the SUT is free from flaws that can be used to subvert the intended security properties of the SUT. Software security testing mainly addresses the *design and implementation vulnerabilities* of the SUT, which is the main focus of the present work.

A. Software vulnerability analysis

A vulnerability assessment is the process of identifying, quantifying, and prioritizing (or ranking) the vulnerabilities in a system [1]. This definition necessarily includes the notion of *vulnerability analysis*, and, more particularly, the challenging issue of finding *exploitable* vulnerabilities. As observed in [2], this process requires in general three main steps:

1) Finding some vulnerable spots in the application, e.g., call to infamous `strcpy()` function;
2) Finding the interfaces for injecting malicious input (taint sources). This is usually derived from some basic knowledge of the functionality of the application, e.g. call to `ReadFile()` or `fscanf()`;
3) Analyzing the application (more often than not at binary level) to find an execution path such that user input can flow to those vulnerable spots.

The last step is clearly the most complicated one, given the complexity and size of the modern applications. A possible approach is to rely on *test-based* techniques, like fuzzing or symbolic/concolic execution. However, as advocated in [3], to be effective in practice, these approaches should not operate on the whole program, but rather on a reduced *test surface* which is likely to contain such dangerous paths.

A natural solution to identify such a test surface is first to (statically) compute some relevant *slice* of the SUT, and then perform the test campaign on this slice. However, as noted in [4], [5], classical static slicing techniques are too heavy to be used for this purpose. One reason is that they are mostly designed to be *complete and sound*, and hence try to compute much more than is needed in this context. If we relax the condition of being *always* safe and sound, scalability issues associated with such approaches can be much better managed [6]. These considerations are one of the starting points of our proposed technique.

B. Objectives and main contributions

In this paper, we embark on measuring the practicability (usability) of static analysis on large applications (specially binary executables of such applications). Static analysis has been criticized earlier for its scalability issues (e.g., in [3], [4], [5], one of the main motivations has been the scalability.), i.e., in case of real-world large applications static analysis may not scale well to produce desired results (this is specially true for classical system dependency graph based analysis [27]). As aforementioned, one of the main reasons is willing to be *always* sound and complete. Therefore, in this paper, we wish to answer the following: *is it possible to make static analysis scalable by compromising its soundness and completeness properties to certain extend such that we can still use it for vulnerability detection on arbitrary application?*

To answer it, we propose a solution to statically compute a set of relevant slices of an SUT for security testing purposes. These slices contain tainted information flows between taint

*This work is partially funded by the BINSEC project (ANR-12-INSE-0002-01).

sources to vulnerable functions, allowing to focus the test effort on program regions which are more likely to be *impacted* by the taint input. This is similar to a dynamic concept called *impact analysis* [7], wherein the whole analysis can be reduced to a (quite small) dynamic slice of the program. Our objective is to apply a similar idea, but from a *static* point of view. This is achieved by considering the *call graph* of the application to capture, at the function call level, the potential interprocedural taint dependencies between "input sources" and "vulnerable functions". The functions within such a slice are subjected to rigorous static analysis to produce relevant *function summaries*, used to confirm/infirm the existence of interprocedural taint dependencies at the instruction level. Having detected the existence of such taintflows, more expensive security testing analysis, e.g., input generation by using symbolic execution can benefit by focusing *only* on such flows. For example, in the context of smart fuzzing [8], [9], we can generate inputs that execute any of the such taintflows, discovered by our technique. This approach is partially inspired by the work described in [10] (use of the call graph).

To be applicable in the context of vulnerability detection, our taint analysis is defined on a binary-level code representation. In particular, this binary code is disassembled and translated into an intermediate representation, called REIL [11]. Binary code analysis raises many specific and challenging issues that are also discussed in the paper.

Our main contributions can be summarized as follows:

- A generic lightweight static analysis framework, allowing to compute information flows between two program locations. The scalability of this approach is obtained using a slicing technique operating at the call graph level, which drastically reduces the set of functions to be analyzed in depth, while keeping under control the accuracy of the results.
- This generic framework is instantiated as a scalable binary level static taint-analysis technique, applied in the context of vulnerability analysis.
- A working prototype called *LiSTT* (**Li**ght-weight **S**tatic **T**aint **T**racer) has been developed.

II. An overview of the proposed approach

As mentioned in the introduction, the main objective of this work is to identify *tainted dependency paths* between pairs of functions calls (IS, VF), where IS is an *Input Source*[1] and VF a *Vulnerable Function*. In this section we illustrate this objective on a simple example and explain our approach in an informal way. We consider the program of listing 1 which essentially consists in reading a file content into a buffer (function `get`), splitting this buffer into a header and a body (function `split`), processing each parts (function `process`), and printing some results (function `print`). For readability reasons we give the source code of this program in C, but our approach proceeds from a (disassembled) binary executable. Although this example is kept simple, it is rather representative of vulnerable programs processing structured files. The code of function `get` is given in listing 2, line 15. This function calls the function `read` (line 17), which in turn calls the function `input` (line 11), to finally calls the library function `fgets` (line 5). `fgets` acts as an **input source** and *taints* its first argument with some user inputs read from an external file. This tainted buffer is propagated back to the `main` (through argument `*b` and then through return values). Hence, variable `buf` of function `main` becomes *tainted* itself (line 3, Listing 1).

```
1 int main(int argc, char *
      argv[]) {
2  char *buf, header[256],
3       *body, *res, *tmp;
4  buf = get(argv[1]);
5  split(buf, header, body);
6  res = process(header,
       body);
7  if (res)
8     print (res, argv[2]);
9  else {
10    tmp = error_msg();
11    print (tmp, argv[3]);
12  };
13  return 0;
14 }
```

Listing 1. Motivating example: `main` function

```
1 void input (char *name,
      char *b) {
2  FILE *fd;
3  fd=fopen(name,"r");
4  b = malloc(1000);
5  fgets(b, 1000,fd);
6     // taints buffer b
7 }
8
9 char *read(char *fname) {
10  char *buf;
11  input(fname, buf);
12  return buf;
13 }
14
15 char get(char *filename) {
16  char *buf;
17  buf = read(filename);
18  return buf;
19 }
```

Listing 2. Motivating example: `get_data` function

```
1 void output (char *name, char *m) {
2  char tmp[256];
3  FILE *fd;
4  fd=fopen(name,"w");
5  strcat (tmp, m);
6  fprintf(f, "%s\n", tmp);
7  fclose (f);
8 }
9
10 void print (char *d, char *fname) {
11  output(fname, d);
12 }
```

Listing 3. Motivating example: `print` function

The code of function `print`, called at lines 8 and 11 by function `main` is given in listing 3 (line 10). This function calls function `output`, defined at line 1. Function `output` contains a call to the **vulnerable function** `strcat` (line 5), those second argument (`m`) is considered as **sensitive** since it may overflow some return address in the execution stack.

Our objective is to identify that this vulnerability is triggered with a tainted value when `print` is called at line 8 of function `main` (Listing 1). Thus, a *potential vulnerable execution path* of the `main` function is $[3, 4, 5, 8]$ (using line numbers of Listing 1). Note that the call of function `print` at line 11 is not vulnerable because the buffer `tmp` is not tainted.

To identify vulnerable paths we need to perform an interprocedural data dependence analysis between input sources and vulnerable functions. However, such an analysis can be time expensive on a large application, especially when conducted on binary code. However, going too deep in a static analysis is not necessarily useful at the whole program level. In this example functions `split`, `process` and `error_msg` do not *introduce* tainted or sensitive data, they can only *propagate* such data through their arguments and return values.

[1] In this context, it is the source of taint input.

As a result they can be abstracted away, and they not need to be included in this analysis.

In the next three sections, we explain the whole approach by formalizing the notions of call-graph slice, data-dependence and summary computations as a data-flow analysis.

III. Proposed Light-weight Data-flow analysis

As stated in the introduction, we wish to make static analysis scalable to get the desired results (in our case, vulnerability detection) by slightly compromising its soundness and completeness in a controllable way. In the following, we will describe the technique that we adopt to achieve this goal.

A. How data flows within a program

We consider programs that are written in procedural oriented programming (e.g. C/C++ programs, compiled to x86 assembly). In such programming paradigm, data can flow from one point to other point by the following two ways:

1) Through function calls wherein data is passed from one function to another via arguments and return value. Therefore, a data point p reaches a function X as one of its arguments from a function Y if Y calls X *transitively*, i.e. if there exists a chain of function calls starting from Y and terminating at X.
2) Through global variables such that a function writes some data into a global variable and later on another function reads data from that global variable.

a) Callgraph Slice: In the above list, we can deduce from point 1 that information can flow from an input source IS to a vulnerable function VF only if there exists a sequence of calls crossing successively IS and VF. Assuming that VF is not (transitively) called by IS[2], such a sequence exists only when there is a "common root" function r (transitively) calling IS and VF. At the call graph level this means that there exists a path from r to IS, and a path from r to VF. We call such path as *Callgraph Slice* w.r.t. r and a pair (s, t) (in the above example, $s = IS, t = VF$) and we denote it as $S_{r \leftrightarrow t}$. The intuition behind this slice definition is to identify the subset of functions involved within an (interprocedural) execution sequence going from s to t. As shown on Fig. 1 (left), $S_{r \leftrightarrow t} = \{r\} \cup C_{r \to s} \cup C_{r \to t}$, where:

- $C_{r \to s}$ (*resp.* $C_{r \to t}$) contains the functions that propagate tainted data "upward" (*resp.* sensitive data "downward") from s to r (*resp.* from r to t);
- $C_{r \twoheadrightarrow s}$ (similarly $C_{r \twoheadrightarrow t}$) is the set of functions indirectly involved in data transmission from s to t: they may introduce *copies* between memory locations.

Example III.1 (Functions outside the slice)**.** *Figure 1 (right) depicts the the call graph of our motivating example (listing 1). The (undirected) path between* `fgets` *and* `strcat` *is shown with thick arrows, and the resulting slice* $\mathcal{S}_{\texttt{fgets} \leftrightarrow \texttt{strcat}}$ *is the sub-graph corresponding to the function set* {`main`, `get`, `read`, `input`, `fgets`, `print`, `output`, `strcat`}*. Functions* `process` *and* `split` *are outside the slice, but they may still propagate tainted values from* `fgets` *to* `strcat`*: a data-flow may exist in* `process` *from* `buff` *to* `header`/`body`*, and in* `split` *from* `header`/`body` *to* `res`*.*

[2]We do not look here for vulnerabilities inside input functions themselves

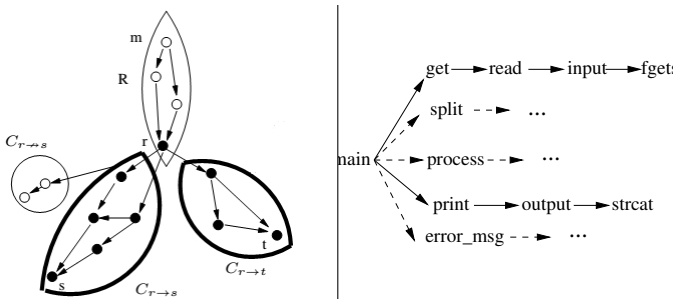

Fig. 1. Call-graph slice: structure (left) and application to Listing 1 (right)

b) Global Variable Based dataflow: If we consider the point 2 in the above list, we can observe that global variables break the callgraph chain and still enabling the data reaching a function. As global variables are visible through out the program, tracing the dataflow through global variables makes taintflow analysis extremely difficult.

B. Scalability Through Compromises

We intend to make analysis scalable by reducing the amount of computation. The main guiding principle is to *compute that is absolutely necessary and approximate the rest*.

c) First Compromise- Soundness: A classical approach to perform dataflow analysis is *data dependency graph* based analysis [12]. However, it computes the expensive dataflow information for every function (intraprocedural) and then proceed to interprocedural analysis. It should be noted that not every function plays a role in propagating the data, but its analysis is performed, anyway. We avoid this by computing expensive interprocedural analysis only for the functions that are included in callgraph slices. For all other functions (e.g. library functions and functions that are called and returned with a function from a callgraph slice), we compute a *default summary*. This default summary is an over-approximation of memory dependencies which means that it may *falsely* state that two memory location (e.g. arguments and return) are dependent on each other, thereby creating a wrong (extra) taintflow. This will introduce unsoundness in the result. However, it may be noticed that *this behavior has not created many false positives* as will be clear in the experimentation part. Note that the accuracy of the analysis can always be improved on demand by computing precise data-flow information for functions outside the slice as well, instead of using default summaries. In case of well-known library functions, we can even replace default summaries by manually created precise summaries.

d) Second Compromise- Completeness: In order to reduce the complexity of the dataflow analysis, we take into account global variables in a limited way: they are considered *only* inside the functions directly calling IS and VF (i.e., their immediate predecessors in the callgraph slice), which is

an under-approximation of taintflow through global variables. The underlying intuition is that, most of the time, if a global variable v is used inside a program to store a tainted data, then v is usually assigned immediately once this data has been read from IS (and at least in the function which called IS). This choice may lead to miss some vulnerable paths, resulting in the loss of completeness of the technique. However, as before, this compromise has not resulted in missing interesting results as is shown in experimentation part. And, here again, a more complete analysis of global variables can still be provided if necessary.

The whole approach we propose to compute taintflow, i.e. potential vulnerable paths) relies on the following steps: (i) for a given pair (IS, VF) create the set of slices $S_{IS \leftrightarrow VF}$; (ii) for each slice S_r in $S_{IS \leftrightarrow VF}$ and for each function f of S_r, perform an intraprocedural dataflow analysis (intraDF) to produce a function summary; (iii) using these summaries, perform an interprocedural dataflow analysis (interDF) to compute the *Data Dependency Sequences* (hereafter called DDS) between IS and VF.

In the subsequent subsections, we provide details for steps (ii) and (iii).

C. Intra-Procedural analysis (IntraDF)

The outcome of IntraDF is a set of *function summaries*, expressing the *effect* of function f in terms of DDS computation. These summaries are used at call sites to propagate the taintflow among functions input/output. The summary of function f gives the *dependence relation* among its input and output sets. It also captures the side effects of a function, telling which of its outputs are *aliased* each others or *tainted*, and which of its inputs are *sentitive*.

Example III.2. *The summary of function* `input` *(Listing 2, line 1) indicates that its 2nd parameter* `*b` *depends on: (i) its 1st parameter* `*name` *(ii) the returned value of the* `fgets` *function (hence it is tainted).*

Simularly, the summary of function `read` *(Listing 2, line 9) indicates that its return value depends on: (i) its input parameter* `fname` *(ii) the 2nd parameter of the* `input` *function (hence it is tainted).*

In the following we use the following notations:

- Tainted(f) (resp. Sensitive(f)) is the set of tainted outputs (resp. sensitive inputs) produced (resp. consumed) by f;
- DDep(f) contains the pairs of inputs-outputs related by a DDS (i.e., able to propagate data).

Summary computation is based on a simple forward dataflow analysis performed on the function Control Flow Fraph (CFG), known as *copy-propagation* (CP) [13]. However, we slightly tweak the classical notion of copy-propagation to fit into our analysis. For example, in the expression $A = B + C$, rather than saying that A is the copy of $B + C$, we interpret it as A depends on B and C. Later if we have $D = A$, we interpret as D also depends on B and C. This dependency relation is confined to arguments, local variables and return value only. Data-flow equations compute, for each statement i and for each register and memory location x accessed in f, a function $In_i(x)$ (respectively $Out_i(x)$) containing the set of registers and memory locations on which x depends before (respectively after) execution of i. Such $In_i(x)$ and $Out_i(x)$ are used to retrieve function DDSs. The remaining questions of how taint and sensitive values are initialized and how do we handle function calls are discussed in the rest of this section.

D. Inter-Procedural Analysis (InterDF)

As mentioned above, function summaries are computed for all the functions[3] of each slice S_r. For other functions, e.g. library functions and those appearing in the sets $C_{r \twoheadrightarrow s}$ or $C_{r \twoheadrightarrow t}$ (see Fig. 1), we use a *default summary* which over-approximates their side effects: DDep(f) $= Input(f) \times Output(f)$, Tainted($f$) $= \emptyset$, Sensitive(f) $= \emptyset$.

Summaries are computed bottom-up on S_r, starting with the functions in the set $C_{r \to s}$ (from leaf nodes to root r), and then with the functions of the set $C_{r \to t}$ (again from leaf nodes to root r).

Within a function f, when an instruction `call g` is reached, then either g belongs to the current slice (and then its summary has been already computed) or we use a default summary for g. If g is a taint source (respectively a vulnerable function), its tainted output (resp. sensitive input) is supposed to be known. This information is then propagated upward using DDep(f).

IV. Implementation and Experimentations

The data-flow analysis described in section III has been implemented within a prototype, called *LiSTT*, operating on binary code to extract a set of potential vulnerable paths. We first discuss some specific issues related to binary level analysis, then we describe the *LiSTT* workflow and architecture, and finally we present some experimental results obtained on real-world applications.

A. LiSTT Architecture

LiSTT is based on both IDA Pro [14], to produce x86 assembly code from binary executables, and BinNavi [15], a framework providing the 3-address REIL [11] intermediate representation together with an API to facilitate data-flow analysis. An example of REIL IR (corresponding to Listing 1) is provided in Listing 4 in Appendix section. LiSTT is written in Python and Jython (∼3500 LOC). Figure 2 provides a high level view of LiSTT, with main steps and components numbered from 1 to 7 to follow the workflow. First, IDA Pro ② takes as input a binary file ① and produces a .idb file which is loaded into the BinNavi framework ③. LiSTT ④ interacts both with IDA Pro (through IDAPython) ⑤ and BinNavi API ⑥ to perform intra- and inter-procedural dataflow analysis. The produced result ⑦ is the set of *vulnerable paths* that have been detected with respect to an input source IS (provided as input). While developing LiSST, our implementation was also

[3] statically linked in the executable

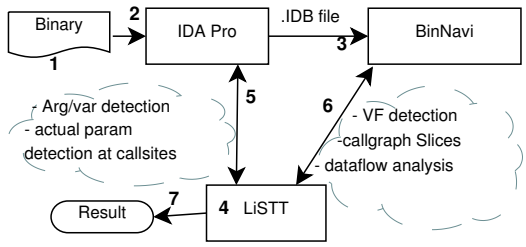

Fig. 2. LiSTT Architecture

driven by the difficulties that are faced when working with binary code. Several of such issues are discussed next.

B. Some Binary Code Specific Issues

As mentioned in the previous sections, we assume that data flow among functions through arguments, return values and global variables. It is therefore necessary to retrieve the addresses of functions arguments and variables in the stack frame (relatively to the `ebp` base pointer on the x86). In our implementation we used the stack frame layouts supplied by IDA Pro [14] using the so-called *semi-naive* algorithm [16].

One shortcoming to this adoption is that we consider a coarse-grained memory model. Although not precise, this algorithm is sufficient for our needs.

Another issue is to identify each callsite and the corresponding actual parameters. Direct function calls are easy to detect. Indirect function call (e.g. `CALL edi` in the x86, where register `edi` contains a function address) are handled using a dedicated dataflow computation (to retrieve the last value assignment to `edi`). More complex patterns like `CALL [edi+offset]` are currently not handled. Regarding parameter passing, we only consider parameters that are pushed onto the stack (i.e. we consider *cdecl and stdcall* only). We also assume that return values are always transmitted through a register (`eax`, in the x86).

C. Experimental Results

The purpose of this experimentation is to show that LiSTT achieves the following objectives: (i) it extracts a *drastically reduced* subset of execution paths from a given binary code; (ii) this subset still contains *existing vulnerabilities*, but it may also allow to detect *unknown ones*; (iii) it can operate on *large applications*. Consequently, LiSTT can be seen as a *flexible tool assistant* for vulnerability detection on large binary application: it can identify possible vulnerable paths while statically analyzing only a restricted part of the code, and the result produced is small enough to be deeply analyzed using more time-consuming techniques. We provide experimental results for (i) the motivating example of section II; (ii) FoxPlayer[17]; (iii) muPDF PDF viewer[18]; and (iv) Serenity Player [19]; and htget (commandline utility)[20]. Each of these applications has known vulnerabilities whose details are provided in Appendix A. Our choice of these examples is based on the previous related works wherein these examples have also been used.

On each of these examples, the input source IS is user-provided, and the set of vulnerable functions (VF) is obtained using the technique[4] presented in [21] (to get a list of *buffer overflow prone* functions).

According to experimental results (Table I), we can see that our proposed approach is able to analyze large applications while producing small enough tainted slices to be analyzed either by hand, or by more computation intensive techniques. Each of these slices corresponds to some traintflow, including the one responsible for the known vulnerability in-hand. Execution times[5] are quite reasonable as well. They mainly depend on the number of slices between IS and VFs (i.e. the size of the set $S_{IS \leftrightarrow VF}$) which, in turn, depends on IS and VFs. Table I summarizes the numerical results we get for each experiment. Columns 1 to 6 give respectively: the application name; the total number of functions (#Func); the number of vulnerable functions (#VF); the number of slices found by LiSTT (#Slices) and the average number of functions per slice (#AF); the number of vulnerable paths (#VP) and the average number of functions they contain; the total execution time (Time).

These results show that the size of the slices extracted by LiSTT (in number of functions) are much smaller than the total number of functions in the whole application. This is particularly true for the largest examples like *FoxPlayer* and *muPDF*. Due to this reduction, in case of *muPDF*, we were able to detect (manually) a vulnerability just on the basis of a vulnerable path calculated by LiSTT. A detailed analysis of the vulnerabilities and LiSTT's detection is presented in Appendix A.

In order to quantify the gain achieved by LiSTT, we resort to a metric called *cyclomatic complexity* (CC) [22]. The CC directly measures the number of linearly independent paths through a program's code [23]. In order to compute the CC for whole program, we use the following formula [24]:
$TCC = \sum_{i=1}^{N} CC_i - N + 1$; where TCC is the total CC of the application, N be the total procedures and CC_i is the CC of i^th procedure.

In our case, each of the slices are considered as *chopped* program and we compute TCC for each slice. Table I (colums 7-10) shows the gain we get in terms of CC: on large applications, the "amount" of execution paths analyzed using our approach corresponds to a very small fragment of the total number of execution paths. A simple conclusion is that LiSTT has far less complexity than that of *program dependency graph* based approaches, thus a far less execution time and better scalability.

V. Related Work

The approach we proposed can be seen as a lightweight interprocedural information flow computation between a source and a destination. It is related both with static information flow analysis and dynamic taint analysis.

[4]Note that other vulnerable patterns could be used as well
[5]on a Dell laptop (2.4 GHz), running Windows XP SP3

Appl.	#Func	#VF	#Slices/ #AF	#VP /#AveFunc	Time	App TCC	Slices		
							Min TCC	Max TCC	Ave TCC (% app TCC)
motivate	30	8	1/6	1/6	6 sec	2	2	2	2(100%)
muPDF	7722	303	47/7	6/4	25 mn	29619	1	1213	276 (0.93%)
FoxPlayer	1074	41	14/8	6/6	33 mn	4784	9	574	119 (2.48%)
Serenity	559	1	1/3	1/3	3 sec	2722	47	845	472 (17.34%)
htget	144	10	5/3	2/3	8 mn	169	1	73	40 (23.66%)

TABLE I
EXPERIMENTAL RESULTS

A. Static information flow analysis

Program slicing (and, to some extend, program chopping) has been intensively investigated. From a static point of view, the most powerful technique proposed is certainly the one based on SDG discussed in [25]. However, building an SDG is quite expensive, and not necessarily required in our context, where we want to favour scalability rather than accuracy. Moreover, only a few works have been dedicated to interprocedural slicing/chopping at the binary level. Such a feature is available for instance in the BAP toolset [26], but only inside a procedure, and an SDG based solution is proposed in [27], but without mentioning any execution time.

As a matter of fact, static *taint analysis* tools are not so popular, as far as the research on taintflow is concerned. An example is the Parfait tool [28], which operates at the source level on large C programs and handles some kind of control-flow dependencies. At the binary level, to the best of our knowledge, the only tool really close to our proposed work is LoongChecker [29], which also provides an interprocedural summary-based taint analysis on binary executables using the REIL intermediate format. However, the tool adopts a rather classic way of analyzing program (i.e. full program analysis), which is arguably not a suitable solution. Our technique is able to produce similar results without being so heavy-weight. Moreover, the tool is not available to do a direct comparison on runtime overhead.

As mentioned earlier in the paper, our work can also be seen as a special application of *program chopping*, introduced in [30] to answer the question of finding "all the program elements that serve to transmit effects from a given source element to a given target element". Reps *et al.* [31] proposed an algorithm for "precise interprocedural program chopping", but this algorithm relies on specific intermediate program representations (PDG and SDG) which explicit all dependence relations between program statements and predicates. Such structures are therefore expensive to compute, and not appropriate for the analysis of large binary applications. The similar ideas were mentioned in a very recent (and perhaps parallel) work by Tripp *et. al.* [5]. The authors of [5] presents a framework called ANDROMEDA for analyzing web application by first constructing a callgraph level slice on demand-driven manner and then performing a fine-grained dataflow analysis on the functions in that slice. The main difference from our work lies in the fact that their work performs taintflow analysis on Java source code. But the message is clear that performing a whole program analysis *a priori* is not necessary for security related analysis.

B. Dynamic Analysis

As mentioned earlier, the problem of taint flow across a binary is well addressed by dynamic analysis of the application [4], [32], [26], [8], [9], [33], [34]. We acknowledge the fact that dynamic analysis of taintflow is very precise vis-a-vis static analysis. However, dynamic analysis can only detect taintflows that appear in the current execution of the application. As far as the objective of our work is concerned, we find the work by Heelan *et. al.* [4] is very close to our work. In [4], the authors maintain the position that their work is to assist the analyst, rather than replacing her and they achieve this by extracting the interesting paths in the binary by doing a dynamic taint analysis and presenting the resulting limited view of the application to the analyst. We have the similar goal in mind while developing LiSTT. Without any further comparison (mainly because ours is a static technique), we briefly summarize few interesting works in this direction.
Dytan [33] is a generic framework for implementing taintflow in binaries. For dynamic instrumentation, it makes use of PIN library and monitors the dataflow across functions by mapping functions arguments and return values. The taint source and taint sink are provided as input to Dytan. It can be observed here that Dytan represents a typical dynamic taintflow analysis framework and most of the tools and techniques for dynamic taintflow follow the similar approach. In [8], the application is instrumented at source code level to monitor the function's arguments and its return values. All such locations that are dependent on the initial input buffer are monitored and when such a location impact the vulnerable function, an alarm is triggered.

VI. CONCLUSION AND PERSPECTIVES

Locating vulnerabilities inside a binary software is still a major and highly challenging software security concern. Since fully automated solutions are not realistic, it is important to develop techniques and tools able to assist human experts in this task. In this paper we proposed a static analysis technique, called LiSTT, to identify program slices containing vulnerable execution paths (i.e., data dependency paths between tainted sources and vulnerable functions). The main driving force in the development of LiSTT is the question that we stared with and we showed that it seems possible to make static analysis scalable and effective by compromising a bit its soundness and completeness properties. We showed on real examples that

this approach scales well, that it retrieves existing vulnerabilities, and that the results produced are small enough to be investigated by hand, or by using more computation intensive analysis techniques. This feature can also helps developers to patch the bug causing the vulnerability. In the context of security testing by means of generating inputs to find bugs by executing the application, LiSTT can be used effectively by reducing the input search space. For example, the program slices produced by LiSTT can be used for evolutionary fuzzing by limiting the genetic algorithm's search space [35], or to guide symbolic/concolic executions towards the most relevant parts of the code (as in [3]).

The prototype we developed is still evolving and there is lot to be done. The complex nature of assembly, specially the ones generated by optimizing compilers, makes binary-level static analysis a difficult process. One of the engineering task in the future will be to add new features, allowing to handle more complex codes and to improve the visualization of tainted paths inside a slice. We also plan to make further experiments with LiSTT. In particular, an interesting issue would be to use this static slice-based approach to detect other typical vulnerability patterns like *use-after-free*.

ACKNOWLEDGMENT

The authors are thankful to anonymous reviewers for their highly useful comments to improve the quality of the paper.

REFERENCES

[1] Wiki, "Vulnerability assessment," http://en.wikipedia.org/wiki/ Vulnerability_assessment.

[2] T. Klein, *Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security*. No Starch Press, 2011.

[3] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of USENIX Security'13*. Washington, DC: USENIX, August 2013.

[4] S. Heelan and A. Gianni, "Augmenting vulnerability analysis of binary code," in *Proceedings of the 28th ACSAC '12*. New York, NY, USA: ACM, 2012, pp. 199–208.

[5] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: accurate and scalable security analysis of web applications," in *Proc. of the 16th international conference FASE*, ser. FASE'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 210–225.

[6] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop PASTE '01*. NY, USA: ACM, 2001, pp. 54–61.

[7] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proceedings of the 25th ICSE '03*. Washington DC, USA: IEEE CS, 2003, pp. 308–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776854

[8] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. of the 2009 IEEE 31st ICSE '09*. IEEE CS, 2009, pp. 474–484.

[9] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[10] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: A control call graph based technique," in *Proceedings of the 12th APSEC '05*. Washington DC, USA: IEEE CS, 2005, pp. 167–175.

[11] REIL, http://www.zynamics.com/binnavi/manual/html/reil_language.htm.

[12] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *SIGPLAN Not.*, vol. 23, no. 7, pp. 35–46, Jun. 1988.

[13] U. P. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[14] Hex-Rays, "Ida pro disassembler and debugger," http://www.hex-rays.com/products/ida/index.shtml.

[15] Zynamics, "Binnavi- binary code reverse engineering tool," http://www.zynamics.com/binnavi.html.

[16] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems*, vol. 32, pp. 23:1–23:84, August 2010.

[17] FoxPlayer Exploit, http://www.exploit-db.com/exploits/11333/.

[18] muPDF, "Vulnerabilities: Cve-2011-0341, 2009-4117, 2009-1605," http://www.mupdf.com/.

[19] Serenity Exploit, http://www.exploit-db.com/exploits/10226/.

[20] HTGET, http://www.cvedetails.com/cve/CVE-2004-0852/.

[21] S. Rawat and L. Mounier, "Finding buffer overflow inducing loops in binary executables," in *Proc of the Sixth International Conference on Software Security and Reliability, SERE 2012*. IEEE, 2012, pp. 177–186.

[22] T. J. McCabe, "A complexity measure," in *Proceedings of the 2nd ICSE '76*. IEEE Computer Society Press, 1976, pp. 407–.

[23] "Cyclomatic complexity," http://en.wikipedia.org/wiki/Cyclomatic_complexity.

[24] "Total cyclomatic complexity," http://www.aivosto.com/project/help/pm-complexity.html.

[25] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up slicing," in *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '94. New York, NY, USA: ACM, 1994, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/193173.195287

[26] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Proceedings of the 23rd international conference CAV '11*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 463–469. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032342

[27] Á. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy, "Interprocedural static slicing of binary executables," in *SCAM*, 2003, pp. 118–.

[28] B. Scholz, C. Zhang, and C. Cifuentes, "User-input dependence analysis via graph reachability," in *IEEE Int. Workshop SCAM '08*, Los Alamitos, CA, USA, 2008, pp. 25–34.

[29] S. Cheng, J. Yang, J. Wang, J. Wang, and F. Jiang, "Loongchecker: Practical summary-based semi-simulation to detect vulnerability in binary code," in *Proc. 10th Int. Conf. on Trust Security and Privacy in Computing and Communications*. IEEE, 2011, pp. 150–159.

[30] D. Jackson and E. J. Rollins, "A new model of program dependences for reverse engineering," *SIGSOFT Softw. Eng. Notes*, vol. 19, no. 5, pp. 2–10, Dec. 1994.

[31] T. Reps and G. Rosay, "Precise interprocedural chopping," in *Proceedings of the 3rd ACM symposium FSE*, ser. SIGSOFT '95. NY, USA: ACM, 1995, pp. 41–52. [Online]. Available: http://doi.acm.org/10.1145/222124.222138

[32] D. Song, D. Brumley, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *In Proc. of the 4th Int. Conf. on Information Systems Security*, 2008.

[33] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proc. of the 2007 Int. Symp. on Software Testing and Analysis*. NY, USA: ACM, 2007, pp. 196–206.

[34] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 2010 IEEE Symposium S&P*, ser. SP '10. Washington, DC, USA: IEEE CS, 2010, pp. 497–512.

[35] S. Rawat and L. Mounier, "An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light," in *Proceedings EC2ND'10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 37–45.

APPENDIX

A. Motivating Example.

In section II, we provided a small hand-written example involving non-trivial dataflow dependencies across functions[6]. An excerpt of its REIL code is shown on Listing 4, and the

[6]About 30 functions in the compiled code.

```
40117400    ldm       eax,    , t0    // 00401174 mov ecx, ds: [
      eax]
40117401    str       t0,     , ecx
40117600    sub       esp, 0x4, qword t0     // 00401176 push
      ecx
40117601    and       qword t0, 0xFFFFFFFF, esp
40117602    stm       ecx,    , esp
40117700    sub       esp, 0x4, qword t0   // 00401177 call
      sub_401024 (get)
...
...
40117F00    add       0xFFFFFFFC, ebp, qword t0   // 0040117F
      mov ss: [ebp + var_4], eax
40117F01    and       qword t0, 0xFFFFFFFF, t1
40117F02    stm       eax,    , t1
40118200    add       0xFFFFFEF8, ebp, qword t0   // 00401182
      mov eax, ss: [ebp + var_108]
......
40118F00    sub       esp, 0x4, qword t0            // 0040118F
      push eax
40118F01    and       qword t0, 0xFFFFFFFF, esp
40118F02    stm       eax,    , esp
40119000    add       0xFFFFFFFC, ebp, qword t0     // 00401190
      mov eax, ss: [ebp + var_4]
40119001    and       qword t0, 0xFFFFFFFF, t1
40119002    ldm       t1,     , t2
40119003    str       t2,     , eax
40119300    sub       esp, 0x4, qword t0       // 00401193 push
      eax
40119301    and       qword t0, 0xFFFFFFFF, esp
40119302    stm       eax,    , esp
40119400    sub       esp, 0x4, qword t0       // 00401194 call
      sub_40110E (split)
40119401    and       qword t0, 0xFFFFFFFF, esp
40119402    stm       0x401199,   , esp
40119403    jcc       0x1,    , 0x40110E
```

Listing 4. Part of the REIL code obtained from listing 1. Text after // shows the corresponding x86 assembly.

resulting slice computed by LiSTT is shown on Fig. 3. As described in section II, the variable `buf` gets tainted through a chain of function calls (through edges 4-0-1-11) and is passed to the vulnerable function `strcat` (through edges 14-8-6). As expected, LiSTT reports that there exists a tainted execution paths from these edges, and that it is here the only one. This example also emphasizes the need for default summaries to correctly take into account functions `split` and `process`, that are outside the slice, but propagate taint information.

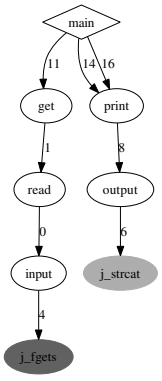

Fig. 3. Taintflow slice for Motivating Example.

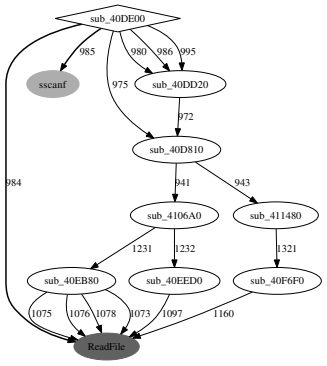

Fig. 4. Taintflow slice for Foxplayer Example

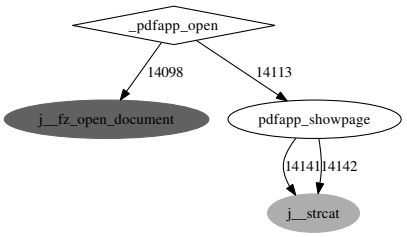

Fig. 5. Taintflow slice for muPDF.

B. *Foxplayer.*

The FoxPlayer version we considered contains 1074 functions, 41 of them being reported as "vulnerable" (i.e., buffer-overflow prone) using the criterion described in [21]. Using function `ReadFile` as the input source, LiSTT extracted from these 41 VF functions a set of 16 slices, and found a tainted path from `ReadFile` to one of the VF in only 5 of them. One of theses slices, shown on Fig 4, corresponds to the vulnerability mentionned in [29]. This vulnerability is a buffer overflow in the function *sub_40DE00* (real name is not known at the binary level). This function calls function `ReadFile` to copy the content of a file (tainted input) into a buffer, and subsequently calls the vulnerable function `sscanf` with this buffer as sensitive argument. This behavior is visible on Fig 4: the flow along the bold edge 984 (from dark gray node to diamond shaped node) introduces the taint, which is passed further along the bold edge 985 (from diamond shaped node to light gray node) to function `sscanf`. The 4 remaining slices could be inspected in the same way to check if they correspond or not to real vulnerabilities. This example shows the ability of LiSTT to find vulnerability in rather large executable while analyzing in depth only a very small amount of code (about 2.5% of the cyclomatic complexity of the whole application, see Table I).

C. *muPDF Viewer.*

muPDF is a light-weight PDF viewer already analyzed in [8], with a few reported vulnerabilities [18]. This is a huge application with almost 7449 functions, 300 of them being considered as vulnerable (VF), according to the criterion given in [21]. Moreover, many functions can be used as taint sources which makes this application hard to analyze manually. In this paper, we choose **fz_open_document()** function as the taint source (IS). LiSTT extracted 71 callgraph slices, with only 5 of them contained a possible taintflow, including the investigated one.

Fig. 5 depicts one of these 5 slices, manually analyzed (both in the source code and using IDA pro). We found that there was indeed a flow of input data to **strcat**, which may lead to a memory corruption. This is a potential candidate for a *new vulnerability* and we are investigating it further to disclose it in responsible manner. In this case, we just have to focus on 4 functions to check this taintflow, which emphasizes how LiSTT can be used as vulnerability analysis assistance tool. The 4 other slices could be investigated in the same way.