DOI: 10.15514/ISPRAS-2019-31(3)-14

Vulnerabilities Detection via Static Taint Analysis

[1] N.V. Shimchik, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>
[1,2] V.N. Ignatyev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
[1] Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. Due to huge amounts of code in modern software products, there is always a variety of subtle errors or flaws in programs, which are hard to discover during everyday use or through conventional testing. A lot of such errors could be used as a potential attack vector if they could be exploited by a remote user via manipulation of program input. This paper presents the approach for automatic detection of security vulnerabilities using interprocedural static taint analysis. The goal of this study is to develop the infrastructure for taint analysis applicable for detection of vulnerabilities in C and C++ programs and extensible with separate detectors. This tool is based on the Interprocedural Finite Distributive Subset (IFDS) algorithm and is able to perform interprocedural, context-sensitive, path-insensitive analysis of programs represented in LLVM form. According to our research it is not possible to achieve good results using pure taint analysis, so together with several enhancements of existing techniques we propose to supplement it with additional static symbolic execution based analysis stage, which has path-sensitivity and considers memory region sizes for filtering results found by the first stage. The evaluation of results was made on Juliet Test Suite and open-source projects with publicly known vulnerabilities from CVE database.

Keywords: static code analysis; taint analysis; vulnerabilities

For citation: Shimchik N.V., Ignatyev V.N. Vulnerabilities Detection via Static Taint Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 177-190. DOI: 10.15514/ISPRAS-2019-31(3)-14

Поиск уязвимостей при помощи статического анализа помеченных данных

[1] Н.В. Шимчик, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>
[1,2] В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
[1] Институт системного программирования имени В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. В связи с большими объёмами кода в современных программных продуктах, в программах всегда существует целый набор малозаметных ошибок или дефектов, которые сложно обнаружить при повседневном использовании или в ходе обычного тестирования. Многие такие ошибки могут быть использованы в качестве потенциального вектора атаки, если они могут быть эксплуатированы удалённым пользователем посредством манипуляций над входными данными программы. В данной работе представлен подход к автоматическому обнаружению уязвимостей безопасности с использованием межпроцедурного статического анализа помеченных данных. Цель данного исследования – разработка инфраструктуры анализа помеченных данных, применимой для обнаружения уязвимостей в программах на языках C и C++ и расширяемой при помощи отдельных детекторов. Этот инструмент основывается на алгоритме решения задачи Межпроцедурных, Конечных, Дистрибутивных Подмножеств (IFDS) при помощи её сведения к специальной задаче о достижимости на графе и способен выполнять межпроцедурный, чувствительный к контексту, нечувствительный к путям анализ программ, представленных в виде LLVM-биткода. Анализа помеченных данных недостаточно для получения хороших результатов, поэтому улучшения существующих методов, мы предлагаем дополнить его ещё одним этапом анализа, который основан на статическом символьном выполнении. Для фильтрации результатов первого этапа выполняется анализ, чувствительный к путям и учитывающий размеры регионов памяти. Оценка результатов была проведена на Juliet Test Suite и проектах с открытым исходным кодом, имеющих подходящие публично известные уязвимости из базы данных CVE.

Ключевые слова: статический анализ кода; анализ помеченных данных; уязвимости

Для цитирования: Шимчик Н.В., Игнатьев В.Н. Поиск уязвимостей при помощи статического анализа помеченных данных. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 177-190 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-14

1. Introduction

In the paper, we consider a specific subset of all possible software vulnerabilities – ones which are caused by utilizing unchecked user-provided data in critical functions or code instructions. This class includes but is not limited to vulnerabilities allowing such important attacks as SQL Injection, Buffer Overflow and XSS attacks.

One group of methods used to represent and discover such vulnerabilities is called taint analysis. In general, taint analysis starts from so-called taint sources – pre-defined functions, which provide special «tainted» data. For example, we may say that the result of a *read()* call will contain untrusted data and thus call it a taint source. Besides that, any value dependent on tainted data is declared to be tainted itself.

There are also so-called taint sinks – special functions or instructions which should never accept tainted data as arguments. Continuing our example, it is not safe to use values, obtained from *read()* call, as a buffer index, since this may lead to a memory corruption and a variety of other problems, thus we may call any pointer dereference instruction a taint sink.

Taint analysis is expected to report such potentially dangerous data flows for a manual or automated verification.

Taint analysis may be performed both as a part of dynamic and static analysis and each approach has its own advantages and drawbacks.

Dynamic analysis is performed during program execution and thus has low false positive rate, but it requires a lot of test runs and it could be close to impossible to explore all possible execution paths in a non-trivial program due to path explosion problem – this is important since some vulnerabilities could actually be hidden on complex execution paths, which are hard to discover using dynamic analysis or testing.

Static analysis on the contrary doesn't execute the analyzed program but processes model instead. Depending on a specific algorithm, this could enable an analyzer to explore almost all possible execution paths, which is a significant advantage in terms of security, but is also likely to increase number of false positives due to inconsistencies between program and its model.

In this work the term "taint analysis" will be used to refer to "static taint analysis" and we define taint sources as all functions providing untrusted data and taint sinks as all instruction parameters or function call arguments which may cause undesired program behavior if one allow an attacker to pick an arbitrary value for it.

We propose some extensions to the interprocedural context-sensitive taint analysis algorithm originally defined in [1]. Implementation of the algorithm is based on the LLVM compiler infrastructure and uses LLVM bitcode as an intermediate representation.

The paper is organized as follows. In Section 2 we briefly discuss the general idea of IFDS algorithm and its application to the taint analysis problem. Section 3 describes several approaches developed

to make memory model used by taint analysis more precise and our improvements of indirect calls resolution. Section 4 summarizes our attempts to decrease false positive ratio by performing additional verification step for all reported vulnerabilities. Section 5 reports experimental results. In the last section, we summarize the results of the work and present directions of future research.

2. Related Work

This section describes the characteristics of the styles used in this document.

2.1 IFDS Framework

Reps et al. [2] introduced an efficient, context-sensitive and flow-sensitive dataflow analysis framework which is able to solve a large class of interprocedural dataflow problems.

This class of problems is called IFDS (Interprocedural, Finite, Distributive, Subset) problems and consists of all dataflow problems in which the set of dataflow facts is a finite set and dataflow functions distribute over the meet operator (either ∪ or ∩). The algorithm solves an IFDS problem in a polynomial time by transforming it into a problem of reachability along interprocedurally realizable paths. The complexity of the algorithm is shown to be $O(ED^3)$ in general case and $O(ED)$ for locally separable problems, where E is the number of edges in the interprocedural control flow graph and D is the number of dataflow facts. According to the algorithm, the program should be represented as a directed super graph $G = (N, E)$, which contains a union of all functions' control flow graphs (CFG) with some special nodes and edges described below.

There is a single entry and exit node for every function in a program.

Every call statement is represented with two adjacent nodes: a call-site node and a return-site node. For every such statement there is an intraprocedural edge from call-site to the corresponding return-site node, an interprocedural edge from the call-site to the corresponding called function's entry node and an interprocedural edge to the return-site coming from the corresponding called function's exit node.

The general idea of the algorithm is to construct a directed exploded super graph $G_e = (N_e, E_e)$ with $N_e = N \times D$ set as nodes (where N is the set of super graph nodes and D is the set of dataflow facts), in which any node (s, d) is reachable from a special start node iff the dataflow fact d holds at node s.

Later several extensions to the IFDS algorithm were proposed by Naeem et al. [3], such as constructing nodes of a super graph on demand (which is important when dealing with large D sets) and exploiting existing subsumption relationships between elements of D set to perform more efficient analysis. It has been reported that these extensions are often necessary when applying the IFDS algorithm to non-separable problems, such as alias set analysis.

2.2 IFDS based taint analysis

Flowdroid [4] is one of the most well-known implementations of the IFDS framework for data leaks detection in Android applications. It demonstrates a possibility to perform taint analysis in terms of IFDS framework and also explains how to combine on-demand backward alias analysis with a regular forward taint analysis. In Flowdroid, dataflow set D is defined as the set of access paths, plus a special «true» fact $[\emptyset]$. An access path consists of a base value (such as a local variable or parameter) with potentially empty ordered list of fields and could be written e.g. like $x.f.g$, where x is the name of the base object, f is the name of the dereferenced field of x object and g is the name of the dereferenced field of $x.f$ object.

Dataflow fact x holds at node s iff an object, which is accessible through this access path at s, may contain tainted data here. x being tainted implies the fact that all object, accessible through this object (such as already mentioned $x.f.g$), are also considered tainted.

3. Taint Analysis Stage

Our experience with the development of vulnerabilities detection tool based on taint analysis shows that resulting warnings contain a lot of false positives. Particular results evaluation makes it possible to discover 2 main roots of the problem: path-insensitivity and inaccurate sizes of tainted objects. It's unclear how to resolve both issues without complex memory model, which would allow to build path and object size conditions. We deal with it by using external symbolic execution engine, forcing it to execute the exploded graph subset corresponding to any specific warning.

Initial warning set is generated by the tool based on [1] and is greatly inspired by Flowdroid design. It uses LLVM as intermediate representation. Due to a low-level nature of LLVM bitcode, we use another definition of access path: an access path consists of a base value (which is an actual LLVM value) and an ordered list of dereference offsets. This definition is suitable for referencing both structure fields (since in LLVM bitcode it is possible to calculate a fixed offset for any structure field) and memory locations, accessible with a help of simple pointer arithmetics. In this paper we will use *[pointer, offset]* to denote value, accessible through dereference of pointer value plus offset bytes, *[integer]* to denote value of the integer, and *[∅]* to denote a special «true» fact. It is possible to specify a list of offsets to define a sequence of consecutive dereferences.

Let's consider an example on fig.1, written in C language.

```
1   extern void *a;
2   extern void *b;
3   void source(int *pointer) {
4       scanf("%d", pointer);
5   }
6   void sink(int size) {
7       memcpy(a, b, size);
8   }
9   void foo(int *t) {
10      source(t);
11      sink(*t);
12  }
```

Fig. 1. Example of a program with interprocedural taint flow

Omitting insignificant details, it is possible to say that

- *scanf* function call on line 4 is a taint source, since it changes the value pointed to by the *pointer* parameter to an arbitrary value chosen by the user;
- *memcpy* function call on line 7 copies *size* bytes from the object pointed to by *b* variable to the object pointed to by *a* variable. We may call it a taint sink, since it is dangerous to specify *size* values greater than actual size of objects pointed to by *a* or *b*;
- *source* function's entry-to-exit subgraph can be summarized with the path edge *(source-Entry,[∅])* → *(source-Exit,[pointer,0])*, i.e. value of *[pointer,0]* becomes unconditionally tainted;
- *sink* function's subgraph can be summarized with the path edge *(sink-Entry,[size])* → *(Sink,[size])*, i.e. tainted data would reach a taint sink if the value of *[size]* is known to be tainted at the entry of the sink function;
- *foo* function's subgraph can be summarized as *(foo-Entry,[∅])* → *(source-callsite,[∅])* → *(source-retsite,[t,0])* → *(sink-callsite,[t,0])* → *(Sink,[size])*, i.e. tainted data unconditionally reaches a taint sink.

Unlike in known implementations and Flowdroid we don't store the whole exploded super graph, because it requires too much memory for regular industrial project with millions of lines of code even if this graph is constructed on-demand. To solve this issue another analysis mode was

developed, which doesn't require exploded super graph edges to be constructed. The existing IFDS analysis engine was supplemented with function summaries (similar to [3]) and explicit taint traces, which makes it possible to show user where the tainted data originate from and how did it get to the taint sink without the need to store the graph itself.

As we noticed during analysis of selected open source projects, taint sources are usually located far from each other in a program and thus their taint flow subgraphs are rarely intersecting. To decrease memory consumption, we have added an ability to run a separate analysis for every taint source. Therefore, exploded graph nodes and summaries can be cleared, but the total analysis time could increase since parts of the program graph can be potentially analyzed more than once.

The second significant difference with other implementations is that each request for a set of aliases for any specific tainted value is handled in a separate local environment with its own IFDS solver and exploded super graph. The graph is cleared after the call and only the resulting aliases set is preserved so that it could be reused both in main taint analysis and when calculating other aliases sets.

Remaining improvements are discussed in following subsections 3.1 and 3.2 with more details.

3.1 Unresolved function calls

When examining a call site, we assume that it is trivially easy to determine the called function by the generated bitcode. Unfortunately, C and C++ programs contain calls, whose targets couldn't be determined until runtime. There are three main sources of such unresolved calls, described below:

1) virtual functions;
2) external function;
3) indirect calls.

C++ virtual function is a member function declared within a base class and overridden by the method in the derived class. When performing a virtual call, the called function is determined by the actual type of the object and may vary in runtime. Such calls were already handled in the previous implementation of the algorithm [1] by adding interprocedural edges to all possible overrides.

Another case of a call with unknown called function is an external call. There are two main kinds of external calls: library functions and system calls. In both cases the analyzed program doesn't contain called functions' definitions and thus we cannot add any "call-site to entry" and "exit to return-site" edges to the call and has to rely on "call-site to return-site" intraprocedural edge only.

By default, we assume that an external function can change values of all its arguments, unless it contradicts with language semantics, so the corresponding facts are not propagated further. We also assume that external function doesn't change value of any global variable and leave it tainted. Usually such assumptions lead to an undertainting and thus we've developed several ways to deal with this issue.

1) Since there is a limited number of system functions and most of them are well-documented, it is possible to create summaries (models) for most frequently used ones manually. Our tool also provides the list of external functions encountered during analysis, which makes it possible for user to prepare summaries for them. It's also possible to specify custom sources, sinks and propagators using similar files in JSON format – those summaries are applied at "call-site to return-site" edges.
2) For an open-source library it is possible to compile it into LLVM bitcode and then link it together with the analyzed program's representation using llvm-link program, which is a part of LLVM infrastructure.
3) If some specific parameter of an external function has type with const qualifier, which specifies that its value should remain unchanged after invocation, we derive a rule for propagating taint through the corresponding argument in every call of this function. Unfortunately, a lot of type-related information, including constancy, may be lost during compilation, so we had to add

several modifications to the Clang compiler in order to store this information in the LLVM metadata.

Lastly, there is another type of calls where the memory address of the called function is calculated at runtime – such calls are named indirect calls. It is possible to say that a virtual call is just a special case of such indirect call. C developers sometimes use indirect calls to simulate virtual calls functionality available in C++ and thus it could be important to support such calls. For example, such technique is used in security-critical OpenSSL library.

We've evaluated several ways to handle indirect calls, described below. Firstly, we make the following assumptions for an indirect call:

a) the indirect call is never used for invocation of any external function, thus the analyzed program contains definitions for all possible candidate functions;
b) the set of all possible candidates is completely determined by the program itself, which means that for all possible target function there is a path in the program where the address of the given target is taken and transferred to the indirect call instruction.

If (A) is considered to be false, such indirect call is actually an external call and should be handled as appropriate, otherwise we can examine following approaches.

1) The naive approach is to take every function definition with compatible parameter types and consider as a candidate. The problem of this approach is that all functions with 0 parameters are indistinguishable and most functions with 1-2 parameters are divided into several large clusters. If we also assume (B) to be true, this approach could be slightly improved by excluding functions, whose address was never explicitly taken in the program.
2) Another approach relies on the assumption that both functions and variables in a program are usually named according to their semantics. In this case it should be possible to compare similarity between call instruction and different call candidates to choose the most likely called function. Unfortunately, the function and variable names are virtually never plain equal and while it should be possible to write an automated heuristic, its results would be unreliable due to lack of formal specifications regarding naming. Such a heuristic would need to put «encrypt string», «EncryptString», «EncryptUTF», «encstr» names into the same similarity cluster, but differentiate between «encrypt» and «decrypt» function names. Common abbreviations, such as «Context – ctx» and «Source – src» may also pose a problem. Right now, we are using a semi-automated solution, where the naive approach is used to generate a .txt file with «expression name» → «{called function names}» mapping, which can be filtered by a user for further analysis runs.
3) Assuming both (A) and (B) are true, it should be possible to implement an interprocedural backward-dataflow analysis to find possible candidates for an arbitrary indirect call. We suppose that one of the problems of such analysis is that its results are used to add missing interprocedural edges to the super graph, but at the same time they are dependent on the super graph structure, thus it should be performed as an iterative process. We don't have a working implementation of this approach by now.

3.2 Memory model

As it was already mentioned in Section 3, we use an access path-based memory model with an access path defined as a combination of base value and an ordered list of dereference offsets.

In the current implementation, offsets are represented with either constant integers or a special λ-offset, which is used to denote an unknown offset. This λ-offset has a special behavior: given any pointer *ptr* and a constant offset *a*, access path *[ptr,λ]* is considered to be subsuming *[ptr,a]*, i.e. the statement `b = ptr[a]` would propagate taint from either one of these access paths to *[b]*, but the statement `ptr[a] = 0` would remove taint from the second one only.

Usage of arbitrary integer offsets instead of object fields in access paths, what is required to achieve complete support of all C++ features, leads to an extremely large *D* set, which has a great impact on worst-case complexity of the algorithm. Thus, it is necessary either to make sure that transfer functions would work with a limited subset of *D* for any given program, or to limit path length in the exploded super graph.

While this special offset enables us to give a simple and efficient representation for complex expressions like `s->packet+len+left`, where `s->packet` is a pointer and `len` and `left` are non-constant offsets, it inevitably leads to an overtaint problem, because λ-offset doesn't restrict the set of possible values and any two λ-offsets are considered to be equal. While it is possible to extend this model to achieve more precise analysis, it's required to keep reasonable size of the *D* set. We've evaluated following approaches.

1) A relatively simple extension of the model is to introduce "unknown non-negative" offset λ+, which is included into "unknown" λ-offset. Let us assume that an instruction writes tainted data into a single element with unknown index of an array field of an object. As a result the whole object becomes tainted, because resulting access path will be equal to [this,offsetof(field) + λ] = [this,λ]. If the used index is nonnegative, because it corresponds to an unsigned variable, it's possible to achieve better precision, tainting only consequent part of the object with the help of λ+. This approach has allowed us to slightly decrease number of false positives on LibTIFF library, but it hasn't proved to make any difference in other cases, since buffers are usually accessed via pointers and this situation seems to be rather an exception.
2) The access path is the core of used memory model. It's possible to use interval domain to get better granularity and precision. The model requires to define several predicates, such as that one access path corresponds to the memory region included into region of another access path. Since access path construction and predicate calculation for interval domain is significantly slower and the total number of created access paths (the size of the D set) grows significantly too, the total analysis time becomes unacceptable.
3) We also tried to implement another extension of memory model which uses symbolic expressions instead of integer offsets in access path. But this approach requires to gather constraints for these offsets using LLVM-instructions which are usually ignored by the IFDS engine, because values of these variables are not tainted. Therefore, it's better to build a dedicated symbolic execution engine and integrate it with existing IFDS engine or to build taint analysis immediately on symbolic execution [5]. Because of this we decided to use an existing symbolic execution tool as a second analysis stage.
4) We also considered using a region-based memory model, similar to the one proposed in [6], since that would allow us to merge all aliases in a single data fact, instead of propagating them independently. We don't have a working implementation of this approach.

4. Analysis Results Refinement Stage

One of the likely reasons of false positives is the lack of path-sensitivity in the IFDS algorithm. Let's consider an example based on a typical buffer overflow test from the Juliet Test Suite for C/C++ on fig.2.

There is a single taint source *fscanf(data)* and a single taint sink *buffer[data]*. Due to path-insensitivity of the algorithm, it reports a dangerous data flow between those instructions, but in reality there is no realizable path from source to sink, because that would require variable *globalFive* to be equal to 5 and not to be equal to 5 at the same time.

There are different ways to solve this issue. We propose performing a two-stage analysis: the first step is done by a relatively fast and simple analyzer, which is able to detect most errors in the program but also produces a high amount of false positives and a second one is performed by a slower but more precise path-sensitive analyzer, whose task is to confirm or reject reports from the first stage.

Similar two-staged approach, consisting of static and guided dynamic analysis was proposed for example in [7] [8]. Preliminary static analysis helps to avoid path explosion problem in dynamic analysis, since it is necessary to check only those execution paths, which were already discovered by the first stage.

```
1   extern int globalFive;
2   int data = -1;
3   if(globalFive == 5) {
4       fscanf(stdin, "%d", &data);
5   }
6   if(globalFive != 5) {
7       int buffer[10] = { 0 };
8       if (data >= 0) {
9           buffer[data] = 1;
10      }
11  }
```

Fig. 2. Example for path insensitivity problem

We propose an analogical combination of two static analyses: IFDS-based analysis and symbolic execution. Unlike building taint analysis using static symbolic execution without IFDS framework, as we have already done for C# in [5], or as it is done for C and C++ in Svace [9], two-staged approach allows to deal with following issues. Every general-purpose static analyzer is required to balance between analysis precision and performance. We can enable very detailed and precise analysis because it's necessary to handle only minimal amount of possible dangerous paths, found by the previous stage. Hence states explosion problem together with conditions simplification for analyzer with states merging are solved.

We decided to use an existing symbolic execution engine for a second stage and the main requirement was that it should work with program representation in LLVM bitcode format to ease exchanging data between stages.

As an experiment, we consider a simpler form of report confirmation – a path confirmation. In this case the only task of the second analyzer is to confirm that the source-sink path is realizable, and it doesn't need to know anything about the vulnerability itself. Hereafter we plan to build more precise condition for each type of detected error. For example, considering usage of tainted data as an array index, we can ensure proper sanitizing by building condition to check if the index is out of bounds.

4.1 KLEE

KLEE [10] is a well-known open-source symbolic execution engine which is actively developed since 2008 and has more than one hundred related publications [11].

It analyses programs in LLVM format and is able to mix both concrete and symbolic execution. To enable symbolic execution, the program should be explicitly annotated with special functions, which are used to mark symbolic values to be created, conditions to be checked etc. This should be done either manually, or by linking the program with a special implementation of system libraries, such as *uClibc* [12].

We have tested a simple way to transfer information about taint sources and sinks in program to KLEE by instrumenting bitcode file with necessary special functions calls.

The path confirmation problem was modeled as follows.

Every warning trace contains an ordered list of instructions (tracepoints) in a program along the path from source to sink, which are important to demonstrate to user the taint flow. For every tracepoint except the last one corresponding to the sink, a special global variable *tracepoint_i_j* is created, where *i* is the index number of the current report trace and *j* is an index number of the tracepoint.

At the first tracepoint of every trace *i* we insert an LLVM instruction, corresponding to the assignment

```
tracepoint_i_1 = 1;
```

At every other tracepoint j of the trace i we insert LLVM instructions, corresponding to the code

```
if (tracepoint_i_(j-1))
    tracepoint_i_j = 1;
```

At the sink we insert an equivalent of the code

```
if (tracepoint_i_(j-1))
    klee_report_error(...);
```

In these terms, the error would be reported iff KLEE has found a realizable path which visits all tracepoints of the trace in a proper order.

Unfortunately, while the concept seemed to be working for simple test cases (and even there were some difficulties with external functions), the general idea has proved to be not so easy to properly implement.

In particular, we encountered following issues.

- KLEE doesn't support memory regions with symbolic size. It means that it's necessary to explicitly specify size for every input string and memory buffer, which is inappropriate for us. This problem is addressed in [13].
- KLEE as is can't start analysis from an arbitrary point of the program. It is acceptable for tests generation, but it is not very suitable for our purpose, since we are interested in simulating of a relatively small subpath from source to sink. In addition, many libraries don't have an entry point at all. The problem is addressed in [14].
- By design KLEE uses program traversing strategy which is aimed to increase code coverage. However, we were not satisfied with the existing "covering-new" heuristic and would need to implement another one for directed symbolic execution similar to [15].
- By default, KLEE works on self-contained isolated programs that don't use any external code (e.g. C library functions), but in practice most programs use external functions calls. To solve this issue, it is possible to link the program with the library or model representation or to automatically generate stub definitions for unknown functions.

After successful experiments on artificial tests, we've tried KLEE to automatically confirm our reports on the relatively small library LibTIFF, but we haven't managed to find a way to reach even the taint source.

As a conclusion, we've decided that it would be too hard to adapt this tool to our problem and it is better to use a static analysis approach for now.

4.2 Svace

Svace [16] is a static analysis tool for bug detection developed at the Institute for Systems Programming, Russian Academy of Sciences. It supports analyzing program written in various programming languages, including C, C++, C# and Java.

Unlike the previous tool, it performs purely static analysis, which means that it doesn't necessarily require a full model of the analyzed program and is suitable to analyze libraries without executable files.

We started with creating a symbolic execution based checker for Svace, which analyses the modified LLVM bitcode file to confirm the existence of a realizable source-sink path for traces of warnings reported by the first stage.

For the proof of concept, report traces are represented in a following manner.

1) Temporarily we don't use any tracepoints, other than the first corresponding to the source and the last one corresponding to the sink.

2) All unique sources appearing in any reported source-sink pair are sorted and enumerated. If tainted data from the source haven't reached any sink, such source is ignored.
3) For every source, a global variable source i tainted is created, where i is the source's index number.
4) A special function call taint_variable(source_i_tainted) is inserted after every source statement in the program to tell the analyzer that the variable is tainted.
5) A special function call check_tainted(source_i_tainted) is inserted before every sink statement appearing in a source-sink pair, where i is the corresponding source's index number.

For every function containing either *taint_variable* or *check_tainted* call, a summary is created. Summary contains intraprocedural reachability condition of the corresponding source or sink. Similar summaries are created for every caller function and contain conjunction of call reachability condition and condition from the callee translated into the caller context. Error condition is equal to the conjunction of the current path condition, the source reachability condition and the sink reachability condition. The resulting condition is passed to a solver for every function call, containing sink. If the resulting condition is UNSAT checker classifies corresponding report as false positive.

Hereafter we plan to check the reachability condition of the whole path or set of paths, instead of source to sink subpath. For example, the entry point can be considered as the entry of nearest function containing source to sink path. We also want to filter out sanitized taints by checking corresponding security conditions.

5. Testing Results

First of all, we performed empirical evaluation of some of the memory usage enhancements mentioned in Section 3. We have launched analysis 4 times on *libssl* library from OpenSSL version 1.0.1f with following configurations:

1) baseline configuration, with most enhancements disabled;
2) current default configuration;
3) default configuration without separate sources analysis;
4) default configuration with full exploded graph instead of currently used taint traces.

This library contains 3 taint sources and was chosen for the demonstration because it contains the well-known «Heartbleed» (CVE-2014-0160) vulnerability, which was successfully found by the analyzer. Also, we have to mention the fact that baseline configuration exceeds 20 Gb RAM usage limit on a full *openssl* executable – it contains 162 taint sources and has a huge taint flow graph mostly because of extensive use of cryptographic library *libcrypto*. Thus, mentioned enhancements seem to be necessary for the analysis of programs with vast taint flow graphs (Table 1).

Table 1. Evaluation of memory consumption

Run	# Reports	# Iterations	Time	Memory
1	75	3 614 thous.	45 s	938 MB
2	75	3 619 thous.	55 s	318 MB
3	75	3 614 thous.	53 s	367 MB
4	75	3 619 thous.	45 s	580 MB

Another launch without «*const*» qualifier handling in external function calls mentioned in Subsection 3.1 showed decrease in amount of reported warnings from 75 to 67, amount of covered functions from 141 to 136 and decrease in amount of algorithm iterations performed from 3.61 millions to 3.02 millions.

Regarding two-stage analysis concept, we performed an evaluation on the set of artificial tests, which was created during development of the first stage analyzer. While these tests were not designed to

test path confirmation, that allowed us to find some obvious implementation flaws and compare analysis time of both stages.

The first stage analyzer has been launched for every test from the set with up to 4 tests being analyzed simultaneously.

On the next run it was supplemented by the second stage analyzer, which has been launched for 176 tests from the set in which first stage analyzer produced at least a single report to be confirmed (Table 2).

Table. 2. Evaluation of analysis time on artificial tests.

Stage	# Tests	# Passed	Time
First	272	269	1m 32s
Both	176	168	18m 36s

Out of 8 tests, incorrectly filtered out by the second stage analysis:

- were caused by the lack of indirect and virtual calls support in the second stage analyzer
- were caused by incorrect interpretation of traces produced by a backward analysis checker
- 2 has failed because of merged or duplicated reports, which seems to be an implementation issue

The substantial slowdown of the second stage analyzer is most likely caused by the fact that Svace is a general-purpose tool and performs a lot of actions which are not necessary for the path confirmation checker. Also, it requires more time to bootstrap and initialize analysis, which makes difference because every test file was analyzed in a separate instance.

We've also checked two-stage approach on Juliet 1.3 test suite for C/C++ [17] with and without work-in-progress Svace checker. The first stage was launched on a program which consists of all unix tests from directories, corresponding to CWE121, CWE122, CWE124, CWE126 and CWE127. There were 2688 taint sources in the analyzed program. Many tests from the set are ignored by the analyzer because they don't contain any taint sources and use a hardcoded invalid index instead.

Then we tried to confirm the results from the first stage with two versions of second stage analyzer: with path confirmation described in subsection 4.2 and another one, which also tries to filter out sanitized taint paths (marked with *, see Table 3).

It should be noted that the first stage analyzer is not yet able to utilize more than a single thread, while the second stage analyzer was allowed to use up to 4 threads during analysis, hence the difference in analysis time and memory consumption.

Table. 3. Evaluation of two-stage analysis on the Juliet Test Suite 1.3 for C/C++.

Stage	# Reports	True positive rate	Time	Peak Memory
First	2424	41%	16m 13s	1.9 GB
Second	2424	41%	13m 10s	8.9 GB
Second*	984	100%	14m 9s	9.8 GB

We don't have a working solution for security conditions checking right now, but for the purpose of proof of concept, we've implemented a simple addition to the current path confirmation checker, which should check that all LLVM values corresponding to the access paths from the reported trace are able to have arbitrary high or negative values within the current data type – otherwise such a taint path is filtered out. This is not a proper implementation of security conditions checking, but is enough to demonstrate filtering out most false positives on this particular test suite.

As it can be seen from the evaluation data, path confirmation checker alone is not enough to improve analysis results on the selected test suite, because it doesn't contain tests with unrealizable paths

between source and sink. However, if supported with a security conditions checker, this approach should be able to significantly decrease number of false positives among reported warnings.

It is also interesting, that while Svace has its own set of buffer overflow and tainted data checkers which cover much greater set of test cases in the test suite, 152 out of our 984 reports seem not to be reported by Svace's own checkers.

We've also tested two-stage analysis on the *libssl* library, which contains "Heartbleed" vulnerability.

First stage analyzer was able to find the taint path from the *BIO_read* call in function *ssl3_read_n* to the *memcpy* call in *dtls1_process_heartbeat*, but also produced more than 70 other reports, which are most likely false positives.

After increasing default limits on procedure analysis time and max annotation size, the second stage analyzer was able to reduce total number of reports to 62 (48), while keeping the true positive report (Table 4).

Table. 4. Evaluation of two-stage analysis on the libssl library from OpenSSL 1.0.1f .

Stage	# Reports	"Heartbleed" found?	Time	Peak Memory
First	73	+	1m 51s	0.4 GB
Second (default limits)	1	-	2m 27s	3.3 GB
Second	62	+	5m 29s	6.6 GB
Second*	48	+	5m 42s	6.9 GB

In case of LibTIFF library and CVE-2018-15209 vulnerability, we could not confirm the true positive taint path with the second stage analyzer, because that would require handling of indirect calls which is not yet supported by the checker.

Therefore, our testing shows that the concept seems to be promising, but still requires further refinement.

6. Conclusion

Performing taint analysis for vulnerability detection via pure IFDS approach has several limitations in comparison to existing buffer overflow checkers, such as [9]: it doesn't make any assumptions about buffer size and is unable to detect several cases even from Juliet Test Suite, because there are no taint sources. For example, if a constant array index is used to access memory outside of array bounds. Moreover, considering error detection problem, pure static taint analysis generates too many alarms to be able to find few vulnerabilities uncaught in an industrial project. The majority of false alarms are introduced by path-insensitivity and overtainting due to inconsistencies between a program and its memory model. Our experience shows that addons to simple and efficient memory model used by IFDS lead to unreasonable analysis slowdown and offer just a minimal results improvement. Therefore, a postprocessing of results is required.

Proposed approach with two-staged analysis looks promising but requires a lot of enhancements to achieve industrial level quality and there are no guaranties that it is even possible.

We are going to continue our research by developing other types of report confirmation in Svace infrastructure, since despite all listed limitations, the tool has a potential to discover serious vulnerabilities, such as «Heartbleed» in OpenSSL and CVE-2018-15209 in LibTIFF.

References

[1] Koshelev V.K., Izbyshev A.O., Dudina I.A. Interprocedural taint analysis for LLVM-bitcode. Programming and Computer Software, 2015, vol. 41, issue 4, pp. 237-245. DOI: 10.1134/S0361768815040027.

[2] Reps T., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In Proc. of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1995, pp. 49–61.

[3] Naeem N.A., Lhoták O., Rodriguez J. Practical extensions to the IFDS algorithm. In Proc. of the international conference on Compiler Construction, 2010, pp. 124–144.

[4] Arzt S., Rasthofer S., Fritz C., Bartel A., Klein J., Traon Y.L., Octeau D., McDaniel P. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 259–269.

[5] Belyaev M.V., Shimchik N.V., Ignatyev V.N., Belevantsev A.A. Comparative analysis of two approaches to static taint analysis. Programming and Computer Software, 2018, vol.44, issue 6, pp. 459-466. DOI: 10.1134/S036176881806004X.

[6] Xu Z., Kremenek T., Zhang J. A memory model for static analysis of C programs. In Proc. of the International Symposium On Leveraging Applications of Formal Methods, Verification, and Validation. 2010, pp. 535–548.

[7] Gerasimov A.Yu., Kruglov L.V., Ermakov M.K., Vartanov S.P. An approach of reachability determination for static analysis defects with help of dynamic symbolic execution. Programming and Computer Software, 2018, vol. 44, issue 6, pp 267-275. DOI: 10.1134/S0361768818060051.

[8] Gerasimov A.Yu. Directed dynamic symbolic execution for static analysis warnings confirmation. Programming and Computer Software, 2018, vol. 44, issue 5, pp. 316–323. DOI: 10.1134/S036176881805002X.

[9] Dudina I.A., Belevantsev A.A. Using static symbolic execution to detect buffer overflows. Programming and Computer Software, 2017, vol. 43, issue 5, pp. 277–288. DOI: 10.1134/S0361768817050024.

[10] Cadar C., Dunbar D., Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proc. of the Proceedings of the 8th USENIX conference on Operating systems design and implementation, 2008, pp. 209–224.

[11] Publications•KLEE. [Online]. Available at: http://klee.github.io/publications/, accessed 20.03.2019.

[12] GitHub - klee/uclibc: KLEE's version of uClibc. [Online]. Available at: https://github.com/klee/klee-uclibc, accessed 02.04.2019.

[13] Šimáček M. Symbolic-size memory allocation support for Klee. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2018. [Online]. Available at: https://is.muni.cz/th/mdedh/, accessed 21.03.2019.

[14] Ramos D.A., Engler D. Under-constrained symbolic execution: Correctness checking for real code. In Proc. of the Proceedings of USENIX Security Symposium, 2015, pp. 49–64.

[15] Marinescu P.D., Cadar C. KATCH: High-coverage testing of software patches. In Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 235–245.

[16] Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignatiev V.N., Zhurikhin D.M., Avetisyan A.I. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, 2014, vol. 40, issue 5, pp. 265–275. DOI: 10.1134/S0361768814050041.

[17] Software assurance reference dataset. [Online]. Available at: https://samate.nist.gov/SARD/testsuite.php, accessed: 20.03.2019

Информация об авторах / Information about authors

Никита Владимирович ШИМЧИК – аспирант Института системного программирования им. В.П. Иванникова РАН. Его научные интересы включают статический анализ программного обеспечения.

Nikita Vladimirovich SHIMCHIK is a postgraduate student of Ivannikov Institute for system programming RAS. His research interests include static analysis of programs.

Валерий Николаевич ИГНАТЬЕВ, кандидат физико-математических наук, старший научный сотрудник Института системного программирования им. В.П. Иванникова РАН, старший преподаватель кафедры системного программирования факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова. Научные интересы включают методы поиска ошибок в исходном коде ПО на основе статического анализа.

Valery Nikolayevich IGNATYEV, PhD in computer sciences, senior researcher at Ivannikov Institute for system programming RAS and senior lecturer at system programming division of CMC faculty of Lomonosov Moscow State University. He is interested in techniques of errors and vulnerabilities detection in program source code using static analysis.