A Meta-level Approach for Multilingual Taint Analysis

Damian M. Lyons* and Dino Becaj*
Fordham University, New York, U.S.A.

Keywords: Multilingual, Static Analysis, Taint Analysis, Software Engineering.

Abstract: It is increasingly common for software developers to leverage the features and ease-of-use of different languages in building software systems. Nonetheless, interaction between different languages has proven to be a source of software engineering concerns. Existing static analysis tools handle the software engineering concerns of monolingual software but there is little general work for multilingual systems despite the increasing visibility of these systems. While recent work in this area has greatly extended the scope of multilingual static analysis systems, the focus has still been on a primary, host language interacting with subsidiary, guest language functions. In this paper we propose a novel approach that does not privilege any one language and has a modular way to include new languages. We present an approach to multilingual taint analysis (a security oriented static analysis method) as a 'meta-level' algorithm which includes monolingual static analysis as a special case. A complexity analysis of the taint analysis algorithm is presented along with a detailed 'deep' multilingual example with Python and C/C++ software. A performance analysis is presented on a collection of 20 public, multilingual repositories selected from github. Our results show an average of 76% improved coverage using our algorithm when compared to monolingual taint analysis.

1 INTRODUCTION

It is challenging to track the flow of sensitive data within a software application to determine if the data is being dangerously exposed (Boxler and Walcott, 2018; Alashjaee et al., 2019). Smartphone apps, in particular, have the potential to expose sensitive personal information, so analysis for Android applications has received early attention (Arzt, 2014). *Taint analysis* methods address this challenge by following the flow of sensitive ('tainted') data in a program to determine if it is being leaked to a 'sink' location without being 'sanitized'. These methods operate either by static analysis of the program (Boxler and Walcott, 2018) or by observing dynamic runs of the program (Alashjaee et al., 2019). This problem is complicated in *multilingual software systems*, defined as systems composed of software written in different computer languages, because data flows are difficult to trace across the language boundaries.

Software developers increasingly leverage features from different languages to implement required functionality more quickly and effectively, resulting in multilingual software systems. The benefit comes at a cost: Mayer et al. (Mayer et al., 2017) reported that 90% of software developers request help with multilingual codebases on issues such as unexpected interactions between languages and security breaches. One reason for this is that cross-language operations are typically opaque. While many IDEs and tools exist to assist a developer in visualizing and analyzing monolingual code, whole program analysis of a multilingual code base is not as well supported. This is especially true for languages such as JavaScript and Python that typically leverage a rich collection of libraries written in other languages (Madsen et al., 2013).

The paper presents a novel approach to the problem of *multilingual taint analysis.* Building on our prior work in analysis of multilingual systems (Lyons et al., 2018; Lyons et al., 2019; Lyons and Zahra, 2020), it proposes a meta-level multilingual static taint analysis algorithm that includes monolingual taint analysis as a special case. There are several ways in which a program in a host language can invoke a function in a different, guest language, and we focus on one of these: the Foreign Function Interface (FFI). Using an FFI, a host language program can call a guest language program as a function call. There may be many different host FFI packages for any two

*The authors are partially supported by grant DL-47359-15016 from Bloomberg LP.

Lyons, D. and Becaj, D.
A Meta-level Approach for Multilingual Taint Analysis.
DOI: 10.5220/0010543800690077
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 69-77
ISBN: 978-989-758-523-4

languages (E.g., Python/C++: PyBoost, pybind11, python.h,...) and more appear regularly. Our proposed solution targets an open-ended aproach to FFI packages and languages by modularizing the monolingual taint analysis step.

The next section reviews relevant prior literature and motivates the proposed approach. Section 3 introduces the approach and presents our primary contribution, the Multilingual Taint Analysis (MTA) algorithm. Section 4 describes the architecture and preliminary implementation of the version of MTA used for performance testing along with a detailed example. Section 5 presents the experimental results from performance testing. Using a sample codebase of public multilingual repositories, we show that MTA improves the coverage, or proportion of suspicious source code opened to inspection, by 76 % on average. The final section summarizes our contributions and presents conclusions and plans for future work.

2 RELATED WORK

Individual software designers may elect to write some parts of their software in one language and some in another but studies indicate that the interaction between languages is often a cause of software engineering concerns (Mayer et al., 2017). Programs in different languages can interact by interprocess communication (IPC), by embedding (e.g., SQL in FORTRAN or JavaScript in HTML), or by employing a host language Foreign Function Interface (FFI), and each has been studied. This paper falls into the FFI category (Furr and Foster, 2005; Lyons et al., 2018; Lee et al., 2020). Since all software ultimately must translate to the machine language for execution, another approach to multilingual analysis is to leverage a common runtime (Grimmer et al., 2018). Our objective is to provide direct feedback as the software is being developed rather than analysis of (possibly binary) executables and thus we employ static analysis of source code rather than dynamic analysis or a common runtime.

Taint analysis can be carried out by dynamic analysis (Alashjaee et al., 2019) or static analysis (Boxler and Walcott, 2018) of the program. Dynamic analysis involves tracing through the program as it executes whereas static analysis evaluates the program without execution. Each approach has advantages and drawbacks: Dynamic analysis is limited to samples of execution but accurately reflects the executions of the program, whereas static analysis covers all possible executions but typically needs to be conservative to address issues of soundness and completeness. Kreindl et al. (Kreindl et al., 2019; Kreindl et al., 2020) present a multi-lingual dynamic taint analysis that is based on their GraalVM runtime. Motivated by the opacity of cross-language function calls (as are we), they support efficient dynamic taint analysis in a selection of dynamic and low-level languages. The implementation leverages GraalVM, a multi-language virtual machine. The architecture employs a language-agnostic core and multiple, language-specific front-ends. Our objective is to develop a static analysis approach to multi-language taint analysis, however, the general structure of (Kreindl et al., 2019) into language-agnostic and langiage-specific modules is something we feel is exceptionally valuable.

It is not uncommon for IDEs to support static analysis in multiple languages (e.g., Eclipse) but it is uncommon to handle whole programs consisting of more than one language. HybriDroid (Lee et al., 2016) can handle mixed Android Java and JavaScript but the approach is not general. (Lee et al., 2020) proposes semantic summaries of FFI calls as a more general framework. (Madsen et al., 2013) use pointer and use-case analysis to analyze JavaScript calls to library functions in other languages. However, the focus in both is on a *primary* host language handling subsidiary guest language functions. A multilingual codebase might contain no primary host language – there may be major components written in each language. Our approach is motivated by this observation to consider multilingual analysis as a 'meta-level' algorithm that incorporates each monlingual analysis as a component.

3 APPROACH

The meta-level approach proposed here is based on two observations:

1. Good quality monolingual taint analysis tools exist and will continue to be developed and improved; and,
2. New FFI packages for multilingual programming are being developed all the time.

The first observation is easily supported. Many capable taint analysis tools are available today: Quandary/Infer (fbinfer.com), Pyre/Pysa (pyre-check.org), FlowDroid (blogs.uni-paderborn.de/sse/tools/flowdroid), SonarCloud (soinarcloud.io) and others. To support the second observation, we need only look at C++/Python FFI packages. The oldest package is the *python.h* FFI, the Python/C API. However, Boost.Python(boost.org) was developed shortly after to offer a more seamless way to call C++ from Python. Subsequently, pybind11 (pybind11.readthedocs.io) was developed to offer the same functionality but in a header-only package. In the case of Java, the basic FFI is the Java Native Interface, JNI. But JNA (github.com/java-native-access) and JNR are FFI packages that each offer an interface improved in different ways.

Taking these two fundamental observations into account, we propose that the best design for a multilingual taint analysis system is one that views monolingual taint analysis and FFI packages as modular components that can be added or removed as needed. We start by introducing some terminology.

3.1 Multilingual Taint Analysis

Let P_ℓ be the compilation unit in source language ℓ that is to be analyzed. Let $ST(P_\ell)$ be the set of statements and $VA(P_\ell)$ be the set of variables in P_ℓ. For convenience we will just use ST and VA when clear.

Some $s \in ST$ may be function calls to a foreign function interface (FFI). If $(a,s) \in FFI_{\ell,\ell'}$ then $s \in ST$ is a foreign function call from host language ℓ to a function in guest language ℓ' with argument a. We only consider the case in which the source for the foreign function call is also available for analysis.

We will assume that for every source language $\ell \in \mathcal{L}$ there is a taint analysis function TA_ℓ defined as follows[1]:

$$TA_\ell(P_\ell, Src, Snk) = L \tag{1}$$

- P_ℓ is a compilation unit in source language ℓ.
- $Src = \{(v,s) : s \in ST, v \in VA\}$ where v in statement s is a source of taint.
- $Snk = \{(v,s) : s \in ST, v \in VA\}$ where v in statement s is a sink to be monitored for taint.
- $L = \{C_i : i \in \{1,\ldots,n\}\}$ is a set of *taint lists* C_i
- $C \in L$ is a sequence (c_j), $c_j = (v_j, s_j), j \in \{0,\ldots,m\}$, where
  - s_{j+1} follows s_j in a possible execution of P_ℓ;
  - if v_j is tainted in s_j then v_{j+1} is tainted in s_{j+1};
  - $(v_0, s_0) \in Src$, each taint list begins at a source; and,
  - $(v_m, s_m) \in Snk$, each taint list ends at a sink.

Using P_ℓ and $FFI_{\ell,\ell'}$ as defined, we apply TA_ℓ with the sink set extended to include all the foreign function calls:

$$TA_\ell(P_\ell, Src, Snk \bigcup_{\ell' \in \mathcal{L}_\ell} FFI_{\ell,\ell'}) = L_{P_\ell} \tag{2}$$

Where $\mathcal{L}_\ell = \mathcal{L} - \{\ell\}$. Some taint lists in L_{P_ℓ} may now end with calls to a foreign function, indicating that taint is being passed to that function through an argument. We propose a mechanism below to continue the taint analysis across the multilingual boundary.

Let $last(C), C \in L_{P_\ell}$ be the final member, c_m, of the taint list C. If $last(C) \in FFI_{\ell,ell'}$, then to further analyze the taint, it is necessary to switch to language ℓ'. However, there is no program $P_{\ell'}$ to analyze with $TA_{\ell'}$. To address this, an artificial *linkage* program is created from $last(C) = (a,s)$ the foreign function call statement s with argument tainted argument a. $Lnk_{\ell,\ell'}(a,s)$ is an artificial linkage program in language ℓ' that declares, sets up the arguments, and then calls the ℓ' foreign function. Analysis of taint continues in ℓ', where $P_{\ell'} = Lnk_{\ell,\ell'}(a,s)$:

$$TA_{\ell'}(P_{\ell'}, Src', Snk' \bigcup_{\ell'' \in \mathcal{L}_{\ell'}} FFI_{\ell',\ell''}) = L_{P_{\ell'}} \tag{3}$$

where

- Src' includes the original argument and call to the foreign function.
- Snk' includes the return from the function and any other arguments (to catch side-effects).

Consider Algorithm 1 which has arguments:

- P_ℓ,
- $FFI = \{FFI_{\ell,\ell'} : \ell, \ell' \in \mathcal{L}\}$ - all available foreign function interfaces,
- $Srcs = \{Src_\ell : \ell \in \mathcal{L}\}$ - all language specific taint sources,
- $Snks = \{Snk_\ell : \ell \in \mathcal{L}\}$ - all language specific taint sinks.

The algorithm must allow for multiple sequential and nested calls to eq.(3) from eq.(2), supporting mutually recursive function calls between languages.

MTA recursively constructs a set L of multilingual taint lists. The **repeat** loop continually evaluates taint analysis until no further items can be added to L. The **for** loop calls MTA recursively for any foreign function call found. If the call propagated taint, then it is added to the list of taint sources and is explored by TA_ℓ on the next iteration of the **repeat**.

3.2 Termination

Let us consider under what conditions MTA will terminate. It is reasonable to assume that there is a finite list of foreign function calls across all the source files available for analysis. Each traversal of the **for** $C \in L$ loop will encounter either new foreign function calls in $last(C)$ or calls already encountered. New

[1] The sanitizer argument is hidden throughout, just for clarity

Algorithm 1: Multilingual Taint Analysis Algorithm.

```
function MTA($P_\ell, FFI, Srcs, Snks$) returns $L$
    $L \leftarrow \emptyset$
    repeat
        $\hat{L} \leftarrow L$
        $L \leftarrow L \cup TA_\ell(P_\ell, Src_\ell, Snk_\ell \bigcup_{\ell'} FFI_{\ell,\ell'})$
        for $C \in L$ such that $last(C) \in FFI_{\ell,\ell'}$ do
            $P_{\ell'} \leftarrow Lnk_{\ell,\ell'}(last(C))$
            $L_c \leftarrow MTA(P_{\ell'}, FFI, Srcs, Snks)$
            for $C' \in L_c$ do
                if $last(C') \notin \bigcup_{\ell'} FFI_{\ell,\ell'}$ then
                    $Src_\ell \leftarrow Src_\ell \cup last(C)$
                end if
            end for
            $L \leftarrow L \cup L_c$
        end for
    until $L = \hat{L}$
    return $L$
end function
```

calls will expand L by $L \cup L_c$; however, previously encountered calls will not, since any taint lists generated will already be in L. Since the list of foreign function calls is finite, and since the **for** loop makes progress through this list each time, ultimately there will be no more new calls encountered. After this happens, L will not increase in size and the **repeat** loop will terminate.

3.3 Complexity

A convenient measure of computational complexity is how often a monolingual taint analysis is conducted TA_ℓ. For worst case analysis, we will assume that taint is found to reach the foreign function call, propagate through it, and reach a sink beyond it. Let n be the total number of foreign function calls and let d be the maximum nesting of calls.

Consider a simple case $n = 1, d = 1$: a program that makes one foreign function call that in turn calls no other foreign functions. The **repeat** loop makes one *TA* call and the subsequent **for** loop makes a recursive *MTA* call to determine whether taint is propagated. The recursive call in this case just needs a single *TA* evaluation to return the tainted result. The foreign function is then added as a source of taint and the loop repeats. The first *TA* evaluation now returns the taint lists from source to the foreign function call, and from the call to the subsequent sink. The prior sequence of *TA* evaluations must repeat one more time before $\hat{L} = L$, calculating a least fixpoint. The time can be written in terms of calls to *TA* as

$$T_n(d) = T_1(1) = 3 + 3T_1(0) = (3+3) \tag{4}$$

The basis case $T_n(0)$ is of a foreign function that calls no other foreign functions, requiring just one call to *TA*, $T_n(0) = 1$. Because there is a foreign function call, taint analysis will still be called but just once. We can generalize the time equation to n calls at every depth (except for the last), and a maximum depth of d:

$$T_n(d) = (n+2) + \sum_{i=1}^{n} (i+2) T_n(d-1) \tag{5}$$

Noting that $T_n(d-1)$ does not depend on i,

$$T_n(d) = a(n) + b(n) T_n(d-1) \tag{6}$$

where $a(n) = n+2, b(n) = \frac{1}{2}n(n+5)$ The closed form can be found by expansion of the geometric series:

$$T_n(d) = a(n) \sum_{i=0}^{d} b(n)^d = a(n) \frac{b(n)^{d+1} - 1}{b(n) - 1} \tag{7}$$

The worst case complexity for MTA is polynomial in n and exponential in d. If we modify MTA so that the **for** $C \in L$ loop never duplicates prior work, then the time complexity drops to linear in n but exponential in d.

$$T_n(d) = (n+1) + nT_n(d-1) \tag{8}$$

If we include that the list of foreign function calls available is finite and an assumption that calls are distributed evenly over the source code then we can approximate a bound on the exponential complexity. Consider that there are a maximum of f foreign function calls and n calls in each monolingual program or linkage program. Taking $b(n) = n$ and one *TA* call for each foreign function call (the 'efficient' case in eq. (8)), then we have that:

$$\frac{n^{d+1} - 1}{n - 1} \leq f \tag{9}$$

Taking the log of both sides and assume that $log_n(n^{d+1} - 1) \approx log_n(n^{d+1})$ and $log_n(n-1) \approx log_n(n)$, we have the approximation

$$d \quad \leq \quad log_n(f) \tag{10}$$

For example, if $n = 20$ and $f = 400$, then $d \approx 2$.

4 IMPLEMENTATION

In this section we describe our intial implementation of Algorithm 1. The implementation currently only supports:

- C/C++ calling Python using *python.h*, and
- Python calling C/C++ using pybind11

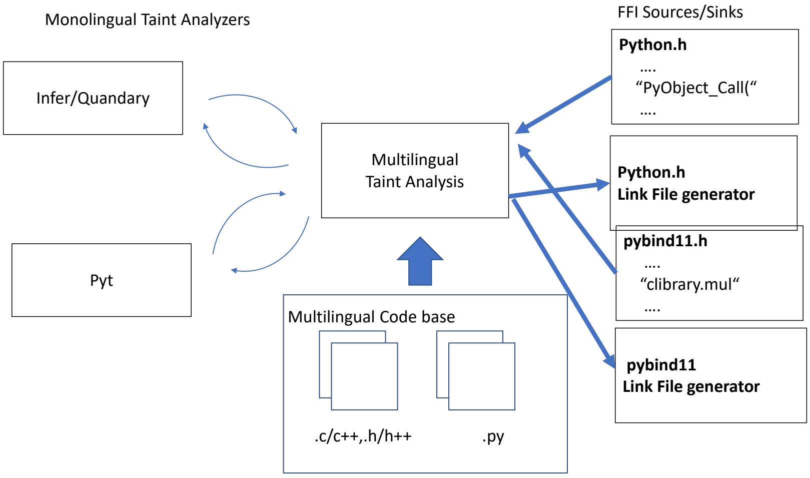


Figure 1: Multilingual Taint Analysis System Architecture.

This decision was made to allow timely performance feedback on the MTA algorithm as there are many public examples of Python/C/C++ code. Adding languages is just a quantitative, not qualitative, enhancement to MTA.

MTA requires that we select a monolingual taint analysis package for each supported language. We elected to use Pyt (Thalman, 2016) for Python taint analysis, due to our prior experience with it (Lyons and Zahra, 2020), and Infer/Quandary for C/C++ taint analysis. For efficiency reasons, the recursive MTA algorithm shown in Algorithm 1 was reimplemented as iterative. Figure 1 shows our system architecture. Each supported language is associated with the following:

1. A monolingual taint analyzer (Fig. 1, lhs);
2. A set of FFI source/sink files and (Fig. 1, rhs/top);
3. A linkage program generator (Fig. 1, rhs/bottom).

4.1 Monolingual Taint Analyzers

The host source language is determined based on the file name suffix. A monolingual taint analyzer is called by MTA for each a source file using a combination of user-supplied sources and sinks and the FFI sources/sinks file. Monolingual taint can be detected between sources and any of the user-supplied sinks as would be expected. However, for any source traced to an FFI sink, MTA begins its multilingual phase.

The linkage program generator is used to build a standalone program in a guest language with a call to the function identified in the FFI sink. The guest language is identified based on which FFI sink was detected. The linkage program constructor also identifies the guest language source file in the codebase and ensures that the linkage program includes the source file as part of its compilation unit.

MTA continues by invoking the guest language taint analyzer on the linkage program in turn. Once the analysis concludes, if taint was passed via the FFI call to any variable involved in the call, then the FFI call is added as a source of taint and the host taint analysis is repeated.

4.2 Foreign Function Interface

The source/sink files for a taint analyzer indicate which source code statements will signal that a variable has become tainted (a source) and which statements are to be monitored for reception of a tainted value (a sink). All FFI API function calls are included with the sinks.

There are two FFI interfaces in this implementation. The python.h FFI is relatively straightforward and includes API calls such as "PyObject_Call" and others. Our main focus is on tracking taint through user code. Some FFI API calls are required to set up the arguments for the FFI, e.g., PyTuple_SetItem. Because we are not interested in tracking taint through system libraries at this point in our research, we added dummy functions for these that always propagate taint.

The pybind11 FFI is a little more involved, since the guest language is hidden in a python module. The existence of foreign function calls is determined by a preliminary pass over the codebase that inspects for the FFI C/C++ syntax. For example, function "bar", defined by a ".def(" pybind11 statement in a file "foo.c++", is written to the pybind11 FFI as "foo.bar". For this initial testing, it is assumed the module will be imported without aliasing.

```
#include ``example.cpp'' // guest fn  defs
 int main(){
   double a1,a2,ret;  // vars for args/reslt
   a1 = taint(a1); // consider a1 tainted
   ret = guestfunction(a1,a2); // function call
   sink(a1);  // is a1 still tainted
   sink(a2);  // was a2 tainted
   sink(ret);  // is ret tainted
   return
   }
```

Figure 2: Example Artificial Linkage Program.

4.3 Linkage Program Construction

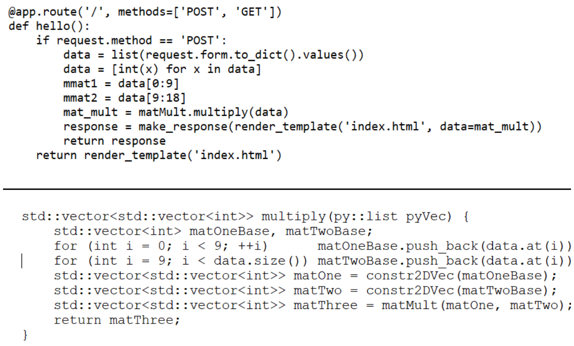
The constuction of the FFI linkage program is similar for all guest languages. The file consists of a main program that includes the guest language source file and calls the guest language function identified from the host language taint analysis. An example is shown in Fig. 2. Any argument to the guest function that was tainted in the host program is specifically tainted by putting them through a "taint()" function in the linkage program. All the arguments and the function return are monitored by sending them to a "sink()" function in the linkage program. The sources/sinks file for the guest taint analysis consists of the following:

1. the "taint(" and "sink(" statements,
2. any user-provided sources and sinks, and
3. the FFI sources/sinks for the guest language.

4.4 Example

An example multilingual system will serve to illustrate the operation of the MTA implementation. The system consists of a web-based interface for a simple matrix multiplication running on a Flask server. A user types numbers into textfields on the Flask-served web page and clicks on a 'multiply' button, at which point the resultant matrix is displayed. The back-end is a Python program 'main.py' that receives the input matrix data from the web page textfields and returns the result matrix for display. However, 'main.py' uses a C++ implementation of matrix multiplication for efficiency. The source code for the Flask back-end is shown on top in Fig. 3 and the C++ multiply on the bottom. The FFI is pybind11.

The sources are the input from the web page via Flask and the sink is the return to the web page via Flask. A monolingual taint analysis would need the source of *matMul.multiply* in Python, otherwise it could not reliably trace taint through the multiply. However, MTA recognizes the FFI as discussed in section 4.2 and calls for a C++ taint analysis. A separate linkage file, test_run.cpp, is generated to include just the call to the resolved FFI call, the C++ *multiply()* function. If analysis of this reports taint propagated then *matMul.multiply* is added as a source of taint and the Python analysis is repeated.



```
@app.route('/', methods=['POST', 'GET'])
def hello():
    if request.method == 'POST':
        data = list(request.form.to_dict().values())
        data = [int(x) for x in data]
        mmat1 = data[0:9]
        mmat2 = data[9:18]
        mat_mult = matMult.multiply(data)
        response = make_response(render_template('index.html', data=mat_mult))
        return response
    return render_template('index.html')
```

```
std::vector<std::vector<int>> multiply(py::list pyVec) {
    std::vector<int> matOneBase, matTwoBase;
    for (int i = 0; i < 9; ++i)      matOneBase.push_back(data.at(i))
|   for (int i = 9; i < data.size()) matTwoBase.push_back(data.at(i))
    std::vector<std::vector<int>> matOne = constr2DVec(matOneBase);
    std::vector<std::vector<int>> matTwo = constr2DVec(matTwoBase);
    std::vector<std::vector<int>> matThree = matMult(matOne, matTwo);
    return matThree;
}
```

Figure 3: Multilingual Taint Example. main.py (top), matMult.cpp (bottom).

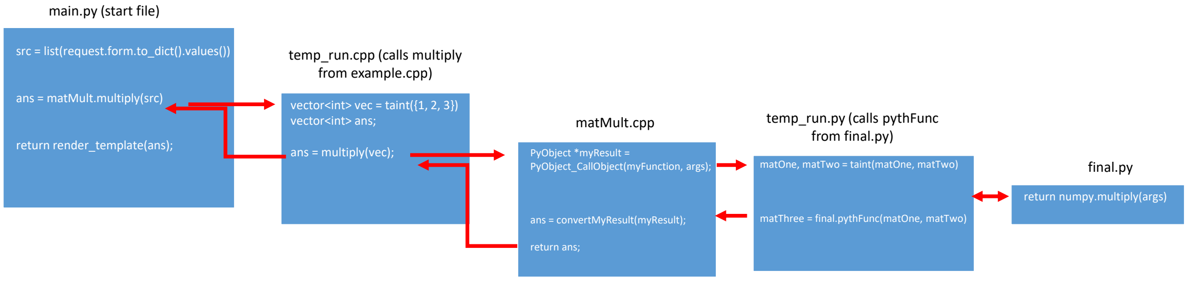
For the purpose of presenting a deeper example, let us consider further modifying multiply.cpp to call a numpy matrix multiplication (instead of the C++ code shown in Fig. 4 bottom) using *python.h*, adding an extra level of multilingual nesting to the system. This modification is shown in Fig. 4 where matMult.cpp now calls a numpy multiply in *final.py*. The analysis of this modified example diverges from the prior example when the linkage file *test_run.cpp* is analyzed by Quandary. An additional FFI call is noted (*PyObject_CallObject*). The linkage file *temp_run.py* is constructed to call the resolved python function *pythFunc* to determine whether taint is propagated. Only if it is propagated will the analysis revert to that of the prior example. The console output from MTA for this analysis is shown in the Appendix.

5 RESULTS

This section presents the results from performance testing of the MTA implementation. It begins by describing how the multilingual programs that comprise the testing codebase were collected and then presents the performance results.

5.1 Multilingual Codebase

The results in this paper were gathered by applying the MTA implementation to Python and C/C++ program which used the python.h and pybind11 FFIs. An internet search was carried out for public github repositories that met these constraints. The approximately 2000 repositories that were returned were further restricted to those that called the guest language


main.py (start file)
temp_run.cpp (calls multiply from example.cpp)
matMult.cpp
temp_run.py (calls pythFunc from final.py)
final.py

```
src = list(request.form.to_dict().values())

ans = matMult.multiply(src)

return render_template(ans);
```

```
vector<int> vec = taint({1, 2, 3})
vector<int> ans;

ans = multiply(vec);
```

```
PyObject *myResult =
PyObject_CallObject(myFunction, args);

ans = convertMyResult(myResult);

return ans;
```

```
matOne, matTwo = taint(matOne, matTwo)

matThree = final.pythFunc(matOne, matTwo)
```

```
return numpy.multiply(args)
```

Figure 4: Multilingual Taint Example: FFI Calls.

Table 1: MTA Performance Analysis Codebase.

Total Num. of Repo.	20
Total Lines of Code (LoC)	1872
Average LoC/Repo.	94
Average Func. Calls/Repo	4.75
Average Forgn. Func. Calls/Repo	2.2
Max Forgn. Func. Calls/Repo	5

software as a function and those for which the guest language function was not a class member (though the remainder of the Python and C++ code could contain and use classes). These were constraints of our implementation. Finally duplicate or overly similar examples were removed. The final codebase included 20 multilingual software repositories.

Table 1 summarizes the 20 repositories. The repositories included a total of 1,872 lines of code in C/C++ and Python. The average repository contained 2 to 3 source files (not including header files) with an average of 94 lines of code. This made them small enough in general to be checked by hand. The largest was >400 lines of code. The average repository contained 2 or 3 foreign function calls and the maximum seen was 5. The vast majority of FFI usage was for Python calling C/C++ (18), the remainder being C/C++ calling Python (2) including 1 repository that had both.

In general the processing of examples was completely automated. For example, the linkage files were automatically generated in every case. However, in some few cases, source files were modified by hand so that source code processing could occur: For repositories that were libraries, a main program was added to call the libary. For repositories that included calls to binary libraries, when it could be done, the library call was replaced with equivalent code. If it could not be done, the repository was dropped from the codebase. We view this manual intervention as a short-term tactic to enable testing and not a long-term characteristic of our approach; we believe these changes, if they continue to be needed, could be automated in the long-term. Finally, as the objective was to evaluate multilingual taint analysis, the main program in each repository was annotated to introduce taint into an input variable that would eventually reach the foreign function call.

Table 2: MTA Performance Results.

Repo. Num.	Mono Taint	Mono Covrg	Multi Taint	Multi Covrg	Multi Impr.
1.	2	8	3	24	0.75
2.	2	6	3	33	0.85
3.	2	16	3	33	0.67
4.	2	8	3	33	0.8
5.	3	27	5	75	0.74
6.	2	6	3	12	0.67
7.	3	30	8	41	0.93
8.	2	10	3	18	0.64
9.	3	15	5	65	0.81
10.	2	6	3	18	0.75
11.	1	4	3	15	0.79
12.	3	18	5	70	0.80
13.	2	6	4	32	0.84
14.	2	6	4	32	0.95
15.	3	18	5	30	0.63
16.	1	3	3	9	0.75
17.	4	52	7	112	0.68
18.	2	8	3	15	0.65
19.	2	8	3	18	0.69
20.	3	18	5	145	0.89

5.2 Performance Results

The MTA performance results on the sample codebase are shown in Table 2. The first column indicates the repository number. The second column indicates the number of taint lists (source to sink) identified with purely monolingual taint analysis. We calculate a *taint coverage* metric as the length of the maximum taint list times the number of taint lists returned. This values both additional taints lists and longer taint lists. The fourth and fifth columns give this data for the multilingual case. Finally we calculate an improvement for multilingual taint analysis over monolingual taint analysis as the multilingual coverage divided by the sum of the mono- and multilingual coverages.

Table 2 shows that the improvement in coverage is 63% or better in all the examples. It reaches as high as 95% for one repository (#14). Multilingual analysis on average improves taint coverage by 76% – showing that it opens up a significant extra amount of suspicious code for inspection.

6 CONCLUSIONS

This research has addressed the issue of taint analysis in multilingual software systems. Although it is increasingly common for software designers to select different languages for different components of a system (JavaScript for the front end, C/C++ for numerical processing, etc.) taint analysis tools are primarily language dependent. In contrast to other other research on multilingual static analysis, our approach does not privilege any language in the codebase above any other. Our proposed solution to extending taint analysis to multilingual systems includes monolingual analysis and cross-language foreign function interfaces (FFIs) as modular components.

The novel contributions of this paper include the proposed Multilingual Taint Analysis algorithm (MTA), an analysis of its complexity and a detailed example, and performance results from our initial implementation of MTA on a codebase of 20 repositories totalling 1872 lines of code. A coverage improvement metric is introduced that reveals how much more source code is opened to inspection for taint using MTA. On average, coverage improvement was 76% for MTA over monolingual taint analysis. The average repository size in this study was relatively small, and this allowed detailed manual checking. However, even with these small repositories, the improvement in coverage is significant. While we believe our results will transfer to larger repositories, that important step is left to future work. Furthermore, the results hold for the C++/Python FFIs covered by the implementation. While we argue that other FFIs can be added with no change to our algorithm design or complexity calculations, supporting evidence for this is also a matter of future work.

The decision to concentrate on C/C++ and Python was made based on the trend to embed C/C++ functions in Python for speed of numerical processing. While MTA clearly adds functionality in this case, we believe that it holds even greater promise for analysis of web repositories which include HTML and JavaScript front ends communicating with backends that may be built in Python. Both Python and JavaScript leverage libraries that are written in other languages. In future work we plan to address this problem using the framework developed here.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the contributions of graduate students Kevin Johns, Courtney King, Arlind Stafaj, Taylor Termine and Benjamin Vecchio.

REFERENCES

Alashjaee, A. M., Duraibi, S., and Song, J. (2019). Dynamic taint analysis tools: A review. *International Journal of Computer Science and Security*, 13.

Arzt, S. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Conf. on Prog. lang. Des. and Impl.*

Boxler, D. and Walcott, K. (2018). Static taint analysis tools to detect information flows. *Int. Conf. Soft. Eng. Research & Practice*.

Furr, M. and Foster, J. (2005). Checking type safety of foreign function calls. *ACM SIGPLAN Conf. on Prog. Lan. Des. & Imp.*

Grimmer, M., Schatz, R., Seaton, C., Wurthinger, T., and Lujan, M. (2018). Cross-language interoperability in a multi-language runtime. *ACM Trans. Prog. Lan. & Sys.*, 40(2).

Kreindl, J., Bonetta, D., and Mössenböck, H. (2019). Towards efficient, multi-language dynamic taint analysis. MPLR 2019, page 85–94, New York, NY, USA. Association for Computing Machinery.

Kreindl, J., Bonetta, D., Stadler, L., Leopoldseder, D., and Mössenböck, H. (2020). Multi-language dynamic taint analysis in a polyglot virtual machine. MPLR 2020, page 15–29, New York, NY, USA. Association for Computing Machinery.

Lee, S., Dolby, J., and Ryu, S. (2016). Hybridroid: static analysis framework for android hybrid applications. *31st IEEE/ACM Int. Conf. on Aut. Soft. Eng.*

Lee, S., Lee, H., and Ryu, S. (2020). Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis. *35th IEEE/ACM International Conference on Automated Software Engineering*.

Lyons, D., Bogar, A.-M., and Baird, D. (2018). Lightweight call-graph construction for multilingual software analysis. *13th Int. Conf. on Software Technologies*.

Lyons, D. and Zahra, S. (2020). Using taint analysis and reinforcement learning (tarl) to repair autonomous robot software. *IEEE Workshop on Assrd. Aut. Sys.*

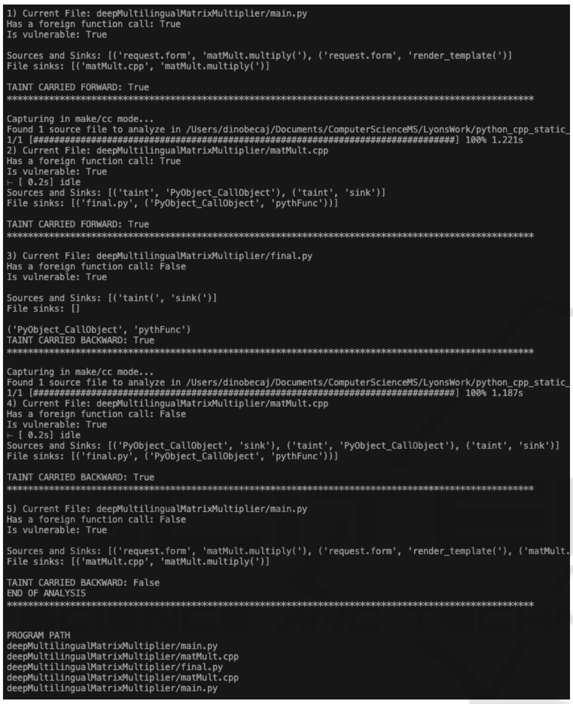
Lyons, D., Zahra, S., and Marshall, T. (2019). Towards lakosian multilingual software design principles. *4th Int. Conf. on Software Technologies*.

Madsen, M., Livshits, B., and Fanning, M. (2013). Practical static analysis of javascript applications in the presence of frameworks and libraries. *9th Joint Meeting on Foundations of Software Engineering*.

Mayer, P., Kirsch, M., and Le, M.-A. (2017). On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development*, 5.

Thalman, B. (2016). Pyt: A static analysis tool for detecting security vulnerabilities in python web applications. *MSC Thesis. Aalborg Univ. Denmark*.

APPENDIX



```
1) Current File: deepMultilingualMatrixMultiplier/main.py
Has a foreign function call: True
Is vulnerable: True

Sources and Sinks: [('request.form', 'matMult.multiply('), ('request.form', 'render_template(')]
File sinks: [('matMult.cpp', 'matMult.multiply(')]

TAINT CARRIED FORWARD: True
****************************************************************************************************

Capturing in make/cc mode...
Found 1 source file to analyze in /Users/dinobecaj/Documents/ComputerScienceMS/LyonsWork/python_cpp_static_
1/1 [################################################################################] 100% 1.221s
2) Current File: deepMultilingualMatrixMultiplier/matMult.cpp
Has a foreign function call: True
Is vulnerable: True
⊢ [ 0.2s] idle
Sources and Sinks: [('taint', 'PyObject_CallObject'), ('taint', 'sink')]
File sinks: [('final.py', ('PyObject_CallObject', 'pythFunc'))]

TAINT CARRIED FORWARD: True
****************************************************************************************************

3) Current File: deepMultilingualMatrixMultiplier/final.py
Has a foreign function call: False
Is vulnerable: True

Sources and Sinks: [('taint(', 'sink(')]
File sinks: []

('PyObject_CallObject', 'pythFunc')
TAINT CARRIED BACKWARD: True
****************************************************************************************************

Capturing in make/cc mode...
Found 1 source file to analyze in /Users/dinobecaj/Documents/ComputerScienceMS/LyonsWork/python_cpp_static_
1/1 [################################################################################] 100% 1.187s
4) Current File: deepMultilingualMatrixMultiplier/matMult.cpp
Has a foreign function call: False
Is vulnerable: True
⊢ [ 0.2s] idle
Sources and Sinks: [('PyObject_CallObject', 'sink'), ('taint', 'PyObject_CallObject'), ('taint', 'sink')]
File sinks: [('final.py', ('PyObject_CallObject', 'pythFunc'))]

TAINT CARRIED BACKWARD: True
****************************************************************************************************

5) Current File: deepMultilingualMatrixMultiplier/main.py
Has a foreign function call: False
Is vulnerable: True

Sources and Sinks: [('request.form', 'matMult.multiply('), ('request.form', 'render_template('), ('matMult.
File sinks: [('matMult.cpp', 'matMult.multiply(')]

TAINT CARRIED BACKWARD: False
END OF ANALYSIS
****************************************************************************************************


PROGRAM PATH
deepMultilingualMatrixMultiplier/main.py
deepMultilingualMatrixMultiplier/matMult.cpp
deepMultilingualMatrixMultiplier/final.py
deepMultilingualMatrixMultiplier/matMult.cpp
deepMultilingualMatrixMultiplier/main.py
```

Figure 5: MTA Taint Analysis of the Deeptaint Example showing repeated calls to Quandary and to PyT.