WHIP: Improving Static Vulnerability Detection in Web Application by Forcing tools to Collaborate

Feras Al Kassar
EURECOM
alkassar@eurecom.fr

Luca Compagna
SAP Security Research
luca.compagna@sap.com

Davide Balzarotti
EURECOM
davide.balzarotti@eurecom.fr

Abstract

Improving the accuracy of static application security testing (SAST) is key to fight critical vulnerabilities and increase the security of the Web. However, even state-of-the-art commercial tools have many blind spots that limit their ability to properly analyze modern code and therefore to discover complex inter-procedural vulnerabilities.

In this paper, we present WHIP, the first approach that enables SAST tools to '*collaborate*' by sharing information that can help them to overcome each other's limitations. Our technique only operates on the application source code by using different tools as oracle to search for signs of interrupted data flows. When we discover such obstacles we inject alternative paths that circumvent the piece of code that SAST tools were not able to handle correctly.

We conducted extensive experiments by analyzing over 100 popular PHP projects with more than 1,000 stars on Github. Our experiments show that our approach enables two popular SAST tools to increase their coverage of the applications' source code, resulting in an increase of up to 25% in the number of high-severity alerts. We manually inspected 30% of the novel 9,226 new alerts obtained by WHIP and responsibly disclosed 35 zero days injection vulnerabilities over 14 applications.

1 Introduction

According to a survey published as part of the *OWASP Code Review Guide* [31], the most common approach adopted by developers to identify injection vulnerabilities in Web applications is through Source Code Scanning Tools. While these tools (also called Static Application Security Testing tools, or SAST, in the industry) are invaluable instruments for vulnerability detection, their accuracy is still fairly limited. For instance, several comparative studies [19, 22, 35] have found that even commercial tools struggle to cope with the complexity of real-world applications. Al Kassar et al. [6] recently studied one of the reasons behind these limitations, by assembling a library of hundreds of PHP and Javascript code snippets (called *testability tarpits* by the authors) whose presence prevented SAST tools from inferring the data-flow link among two elements of a target program.

One of the main findings of Al Kassar's study was that tarpits affect different tools in different ways: what poses a problem from one tool may be analyzed correctly by another and vice versa. Moreover, the authors noticed that these code patterns are very common in today's applications, with the average project on Github containing 21 different tarpits, each present multiple times. This translates to the fact that even the most advanced commercial SAST tools on the market were unable to analyze applications in depth, without encountering a pattern that prevented them from correctly modeling the code [6].

These limitations are well known by practitioners, who try to mitigate the risk of false negatives by analyzing their application with multiple static analysis tools, in the hope that what a product misses, another can find. For instance, the NIST organization published a document on static code analysis [25] where they explicitly suggest the best practice of combining the results of two or more tools.

This idea of combining the alarms generated by different static analysis tools is also often supported by researchers. For example, Nunes et al. [28] performed an empirical study of combining the results of static tools. Muske et al. [27] published instead a survey about research directions on handling static analysis alarms. The authors cite many papers that discuss the concept of alarms ranking, where the severity of an alarm is chosen based on how many tools raise the same alert.

Unfortunately, combining the alarms of different tools can reduce the risk of false negatives only to a certain extent. In fact, any sufficiently complex application would contain enough different tarpits to impede the analysis of all SAST tools. Thus, even if for different reasons, it is very likely that each tool would encounter a snippet of code it cannot handle correctly. For this reason, in this paper, we argue that it would be more beneficial to somehow combine the *internal model* reconstructed by different tools, and not just the vulnerabilities they discover. However, the collaboration between the tools – where one tool uses its strength to overcome the weaknesses of another – has been rarely explored by researchers. NAVEX [8] only mentions the collaboration between static and dynamic analysis solutions, when a crawler (dynamic) can be used to retrieve the flow between files that are later analyzed by a static tool.

In this paper, we present the first approach **to enable the collaboration between SAST tools**. Our novel technique only operates on the application source code, thus allowing our approach to be applied also to commercial tools, without the need to access their internal data structures. Our idea is to search for signs of interrupted data flows, by using the tools as oracles, and then inject another path to circumvent the piece of code that tools were not able to handle correctly.

Our approach is general and can be applied to any programming language. As an example, we implemented a fully-automated prototype, called WHIP, targeting the PHP language, which today is still by far the most common language to develop Web applications (78% market share in 2022 [1]).

We conducted a number of experiments to show that WHIP can increase the amount of source code processed and in turn lead to a higher number of security alerts. For our experiments we used two research tools (WAP and Progpilot) and two commercial SAST tools (Comm_1 and Comm_2) [1]. Research tools did not show any benefits in the experiments due to the stronger effectiveness of commercial tools compared to the research tools. Basically, no data flow paths were emerging from research tools that were not already discovered by commercial ones. For this reason, we focused the rest of our experiments on the commercial tools and provide additional experiments targeting research tools in Appendix A. Over 114 popular projects (all with more than 1,000 stars on Github) Comm_2 reported 25% and Comm_1 reported 10% more alerts, corresponding to 9,226 new high-severity alerts that none of the tools was able to discover in isolation. By sampling and manually investigating 2,732 (30%) of these new alerts, we confirmed the discovery of 35 zero-day vulnerabilities across 14 applications, 24 of which have already been confirmed by the respective developers. However, research tools did not demonstrate clear benefits in comparison to commercial tools in our approach.

Finally, we compared the complexity of the new vulnerabilities discovered by WHIP with a dataset of 100 CVEs. Our analysis shows that by using a tool to overcome the limitations of another, both tools are able to explore deeper into the target dataflow. For instance, while the average vulnerability in previous CVE contained a path (between a source and a sink) of only 7.8 lines of code (LOC), the shortest path among our 35 new discoveries is 12 LOC long, and the average is 25.

The rest of the paper is organized as follows. Section 2 presents background information on the static analysis tools and their efficiency in detecting injection vulnerabilities. We then present a motivational example (Section 3) inspired by one of our discoveries. Section 4 illustrates our approach and the algorithm that we created to apply the changes to the source code and force different tools to collaborate. We discuss the impact of false positives and false negatives in Section 5. Finally, we present the design and results of our experiment in Section 6 and 7.

2 Background

Static analysis tools scan applications without the need of deploying the project, by analyzing their code for signs of security issues [23]. Researchers have proposed different models to capture both the syntax and the semantics of source code. *Code property graphs* [39] (CPGs) became one of the most popular by merging in a single model the abstract syntax tree with three other graph-based representations: the control flow graph to represent the order of execution of the statements, the program dependency graph to capture the dependency between two statements, and the call graph to represent functions and methods invocations.

This graph-like representation is particularly suited to detect one of the most prevalent classes of vulnerabilities, called *injection vulnerabilities*. Injection vulnerabilities occur when an attacker can inject harmful values into an application that lead to unexpected results when interpreted by other parts of the system. For example, an attacker-controlled snippet included in an HTML page can lead to an XSS vulnerability, which can cause the victim browser to execute attack-provided code.

To detect these bugs, SAST tools need to reason about the flow of user-provided information through the

[1] For legal reasons, we have to anonymize commercial products.

program, starting from the points where attackers can inject their input (called "*sources*") until the points in the program where this input is consumed and interpreted (called "*sinks*"). The variables which carry the data between sources and sinks are called "*tainted variables*". Thus, detecting injection vulnerabilities boils down to discovering a data-flow path that connects a source to a sink, along which the data is not properly sanitized.

This process presents two main challenges. First, the static analysis tool needs to be able to construct the path in the first place, by understanding how data can propagate among different variables and different parts of the code (a process normally called taint propagation). Second, the tool needs to correctly analyze the resulting path to detect whether the user-provided input is properly sanitized to protect against the specific type of injection vulnerability that has been considered. Errors in these two steps can cause the tool to miss vulnerabilities, but also to raise false alarms.

To capture the reason behind these errors, Al Kassar et al. [6] recently proposed the concept of the *testability tarpits*, i.e., specific code patterns that can prevent a static tool to properly analyze its code (and therefore build a correct internal model). The authors found hundreds of these tarpits and showed that even the best SAST tools are strongly impacted and cannot fully analyze real-world applications.

Finally, even if the internal graph representation of a web application is built correctly, it is often very large and very time-consuming to explore exhaustively. Thus, static analysis tools often employ several thresholds to limit their analysis and produce results in a reasonable time. For example, in our experiments, we noticed that commercial tools apply thresholds (which are often outside the control of the user) to limit the depth of the call graph, as well as the length of the data and control flow paths they analyze.

Some testability tarpits could be mitigated by modifying the tool and improving its code analysis engine. This can be challenging in the case of commercial tools or when research tools are no longer supported. On top of that, not all testability tarpits can be resolved. For instance, to handle dynamic features (such as reflection and dynamic function invocation) static tools can only offer solutions through over- and under-approximations: the first increasing source code coverage at the price of higher false positives, the second ignoring certain features at the price of increased false negatives. Each commercial static tool is optimized to find the right balance between accuracy, the number of alerts provided to developers to manually review, and the time and resources required to scan projects. This tuning requires static tools to carefully choose the type of code analysis they implement and the threshold they use to control their operation.

As a result of all these limitations, even state of the art SAST tools are often limited to the discovery of *shallow* vulnerabilities.

```
1  <?php
2  function func1($vars){
3      $res = "";
4      foreach ($vars as $var => $val) {
5          $res = $res . $var;
6      }
7      return $res;
8  }
9  function func2(){
10     $args = func_get_args();
11     $ret = call_user_func_array('sprintf', $args);
12     return $ret;
13 }
14
15 $vars = $_POST;
16 $x = func1($vars);
17 $y = func2($x);
18 echo $y;
```

Listing 1: Example of an XSS vulnerability

3 Motivation

Listing 1 shows a snippet of PHP code inspired by a real XSS vulnerability we discovered in the Cacti fault management framework. The vulnerability exists because the attacker controls the `$_POST` variable at line 15, whose value can reach, without being properly sanitized, the `echo` statement at line 18. Despite the fact that the code is very simple, none of the SAST tools we used in our experiment can detect this vulnerability.

To understand the reason we can look at Figure 1, which shows the data flow graph of our motivational example. In the figure, there are three sets of blocks, associated respectively to the main function (in the middle) to `func1` (on the right), and to `func2` (on the left). Finally, the edges illustrate the data flow between the different lines of code.

In general, SAST tools identify vulnerabilities by detecting a path between a source and a sink [9, 39]. However, in our example, none of the SAST tools that we tested, including the commercial ones, encountered a specific testability tarpit that prevented them from detecting the data-flow that connects the source to the sink. We can detect the testability tarpit that the SAST tool couldn't "understand" by transforming the corresponding line of code and test again if the tool can detect the vulnerability. If it detects it in the second

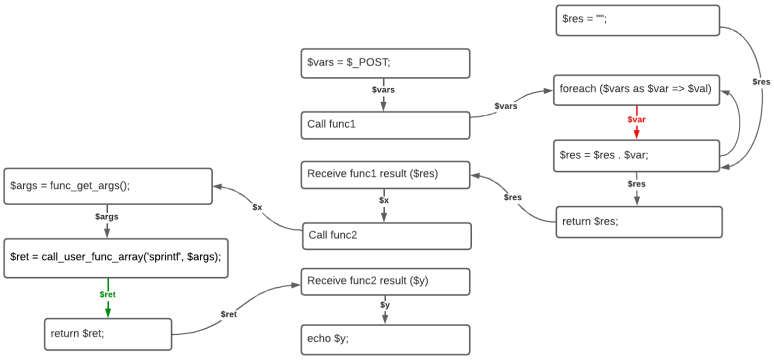

Figure 1: The data flow of the motivational example

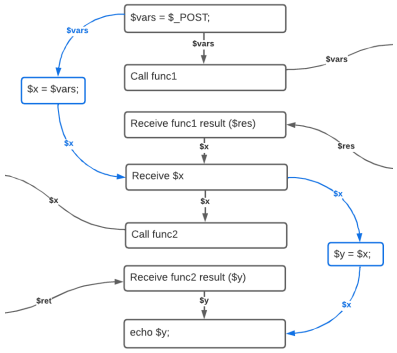

Figure 2: The motivational example with the solution

case but not in the original we can conclude that it is unable to infer that particular edge. In one case, `func1` uses a `foreach` loop to concatenate the keys of an array, an operation that is not handled correctly by Comm_1. As a result, the red edge in Figure 1 would be missing from Comm_1 internal representation, thus breaking the path associated with the vulnerability. The second case is due to `func2`, which receives its arguments through the built-in function `func_get_args` and then calls the built-in function `sprintf` dynamically through the PHP function `call_user_func`. This code poses problems to Comm_2, resulting again in a missing edge (the green one in the figure) and thus in a fragmented path that prevents the tool to detect the vulnerability.

This example clearly shows two important aspects, which served as motivation for our research. First, the fact that the reasons why SAST tools fail to discover vulnerabilities are different [13, 21, 36]. Second, the fact that existing tools cannot be *combined* to overcome each other's limitations. Today, all an analyst can do is run both tools in isolation, in both cases failing to discover the aforementioned vulnerability.

Thus, the main goal of our research is to propose a new way to allow a tool to help another. Our intuition is that if we could transfer somehow the part of the dataflow graph of `func1` from Comm_2 to Comm_1, then Comm_1 would have a complete picture of the program and could detect the unsanitized data-flow path associated to the XSS vulnerability. Similarly, transferring the missing red edge from Comm_1 to Comm_2 would achieve the same result, this time helping Comm_2 to detect the vulnerability. In other words, while different tools use different strategies and are affected by different limitations, their *combined model of the program is more complete, and therefore more effective at finding bugs, than their two models in isolation*.

4 Approach

As we explained in the previous section, our goal is to share information about the internal models of two or more tools. However, many popular SAST tools are commercial applications that do not provide access to their code or data structures. Therefore, we need to find a general solution that considers each tool as a black box, which poses a serious constraint to the design of our system.

Our solution is to operate only on the source code of the target application, by using an approach based on two main operations: *infer* and *stitch*. The *infer* operation extracts security-relevant data-flow paths (i.e., those that originate from a source that can contain user-controlled input values) from one SAST tool. To achieve this, we first inject fake sink instructions related to one type of vulnerability (e.g., `echo` for XSS) into the target application. Then, we scan the modified application with each SAST tool and we process all the reported alerts related to the fake sinks. If the path between a source and the fake sink is reported as vulnerable by tool *A*, it means that the tool was able to build an uninterrupted data-flow path between the two statements. Then, if the same path is *NOT* reported as vulnerable by another SAST tool *B*, we can deduce that its code likely contained some testability tarpits that prevented *B* to reconstruct the same data-flow. Thus, we can use what we learned from *A* to "*stitch*" (the second building-block of our approach) the two ends of the data-flow together, by creating a new edge in the data-flow graph that can help *B* to conduct its analysis.

It is important to note that we do not permanently modify the application. Instead, our technique only

Algorithm 1 Approach

```
1:  Input
2:      A     web application
3:      Ts    set of SAST tools
4:      V     vulnerability type
5:  Output
6:      Ns    new findings of type V
7:  Prepare
8:      i ← 0
9:      ST^i ← ∅
10:     Fs^i ← scan(Ts,A,V)
11: InferStitch
12:     A^i ← inject(A,V)
13:     repeat
14:         i ← i+1
15:         Fs^i ← scan(Ts,A^{i-1},V)
16:         ST^i ← infer(A^{i-1},Fs^i)
17:         A^i ← stitch(ST^i,A^{i-1})
18:     until ST^i ≡ ∅
19: Evaluate
20:     A^i ← clean(A^{i-1})
21:     Fs^{i+1} ← scan(Ts,A^i,V)
22:     Ns ← diff(Fs^{i+1},Fs^0)
```

creates a temporary variant for the purpose of SAST testing. While the variant does not preserve the original semantic of the code, it does not need to be executed and its only purpose is to increase the code coverage of SAST tools and their ability to discover vulnerabilities.

In summary, our technique modifies the application by using a set of SAST tools as oracles to infer which variables are connected to a given source without being properly sanitized. If a tool detects these connections, we forcefully add new data-flow edges to the application (through new variable assignments) to help other tools discover the same connections. The implementation of our prototype is available in our repository [4].

In order to implement this idea, we first need to decide at which granularity we want to perform our "infer and stitch" operations. For instance, one could implement this approach at the variable assignment level, adding a fake sink every time a variable assignment takes place. However, such a fine-grained solution would require the introduction of a huge number of fake sinks and, as a consequence, a very long post-processing phase to analyze the SAST findings.

Therefore, we decided to implement our solution at the function level, where information can only flow between parameters or from a parameter to a return value, thus limiting the number of fake sinks we need to introduce and the required SAST processing time.

ß The complete approach is presented in Algorithm 1. It takes as input a web application A, a class of vulnerabilities V, and a set of SAST tools Ts. It then outputs any new alert (of the selected vulnerability class) generated from the SAST tools after our transformation. We now explain our approach in more detail, by using again our motivational example presented in Listing 1 and two commercial tools.

4.1 Phase I: Prepare

In this phase (cf. lines 7-10 of Algorithm 1), we simply init two variables (i.e., the iteration step i is set to 0, the initial set of stitches ST^0 is set to the empty set) and we run the SAST tools against the original application to collect the set of findings Fs^0 of type V (e.g., XSS). These findings will serve as a baseline to evaluate the effectiveness of our approach in detecting novel findings. When running Comm_2 and Comm_1 on our motivational vulnerable example, Fs^0 is empty as both tools are unable to discover the XSS vulnerability when used in isolation.

4.2 Phase II: Infer and Stitch

In this phase (cf. lines 11-18 of Algorithm 1), our first objective is to use a set of SAST tools to determine if there is propagation of tainted values between the input and output of functions. To do this, we modify the application source code by "injecting" (cf. line 12) fake sinks that are related to the chosen vulnerability type after each function call. These sinks should be recognizable by all the static tools in the approach. For example, we choose to use the `echo` statement to write a code snippet that directly prints the user-provided input. We then scan this code using our tools and if they all detect the presence of an XSS vulnerability in the code, we consider the sink to be suitable for our approach.

Our tool uses this technique to print both the inputs and output values of each function. Note that, no fake sinks are added for functions that neither return a value nor get input parameters. Since printing a variable can lead to an XSS vulnerability, we expect SAST tools to raise an alert if the variable is tainted (i.e., it can contain unsanitized user input).

For instance, in our motivational example our approach would modify the call to `func1`, by adding two fake sinks, as follows:

```
$x = func1($vars);
/*E1-16*/ echo $vars;
/*E2-16*/ echo $x;
```

These added instructions are labeled with `E1-16` and `E2-16` to indicate they are fake sinks generated for the function call at line 16 of Listing 1.

The approach is now ready to iterate over the infer and stitch operations (cf. `repeat-until` loop). Once the

iteration step is increased (cf. line 14), the SAST tools are run against the modified application (cf. line 15) and the `infer` operation (cf. line 16) is then used to process the SAST findings. In particular, the SAST findings related to the injected fake sinks are automatically inspected and the following conditions evaluated:
[C1] at least one SAST tool (say *A*) detects that tainted data can flow from one function parameter (say `$in`) to the return value of the function (say `$out`); and
[C2] another SAST tool (say *B*) reports that tainted data can only flow into `$in`, but not to `$out`.

If both **[C1]** and **[C2]** are true, our approach uses the findings of tool *A* to help tool *B* by introducing a new "stitch" in the dataflow to enforce the data-flow connection between `$in` and `$out`. The inferred stitches are concretized in our approach again as a modification at the source code of the application, this time through the use of a conditional assignment for each stitch. This is done by the `stitch` operation at line 17 of our algorithm. For instance, for the stitch `ST1` capturing the data-flow connection between `$in` and `$out`, the conditional assignment hereafter would be added just after the function:

```
$out = func($in);
/* STITCH_BEGIN: ST1 */
if(round(rand(0,1))){
  $out = $in;
}
/* STITCH_END */
```

Listing 2: Simple stitch

We wrap the assignment inside an `if` statement to create an alternative edge in the data-flow, without completely replacing the path through the function. This, as we will explain in Section 5, is important to prevent our transformation to introduce new false negatives. As a condition we chose an expression that is randomly computed as true or false at runtime.

In the general case in which a function has more than one parameter, they are all individually tested and, if more than one argument is part of taint propagation, multiple edges (stitches) will be introduced in separate conditional blocks – like in the following example:

```
$out = func($in1,$in2);
/* STITCH_BEGIN: ST1 */
if(round(rand(0,1))){
  $out = $in1;
}
/* STITCH_END */
/* STITCH_BEGIN: ST2 */
else if(round(rand(0,1))){
  $out = $in2;
}
/* STITCH_END */
```

Note that while adding stitches, the fake sinks of the input parameters that were used to infer these stitches are removed, as they are not needed anymore. If the fake sinks of all the input parameters of a function are removed, then also the fake sink of the function return variable is removed.

We now describe the whole iterative approach against our motivational example. In the first iteration, Comm_2 raised an alert for both `E1-16` and `E2-16`, a sign that its static analysis algorithm correctly concluded that `func1` propagated tainted information from its parameter to its return value. However, while Comm_2 succeeded, other tools might miss this connection. In fact, in this case Comm_1 raised an alert for `E1-16` but NOT for `E2-16`, due to its inability to process correctly the `foreach` loop in the function body.

To sum up, from this first iteration our approach learned that, through the function `func1`, tainted data propagates from the `$vars` variable to the `$x` variable. Since not all SAST tools in our set detected it, our approach forcefully add this dependency in the program.

To make the relationship between the `$vars` and `$x` variables explicit, our approach modifies again the source code of the application, this time by adding a simple conditional assignment as in Listing 2 where `$vars` and `$x` replace `$in` and `$out`, respectively.

In summary, with the `stitch` operation of our approach, we modify the application to add new instructions that explicitly connect two variables, when at least one tool detects that tainted data can flow from one to the other.

It shall be noted that one iteration of the infer and stitch operations is not sufficient to discover the vulnerability of our motivational example. In fact, if we consider the whole code of our example, during the first iteration Comm_2 raises alert for both the input and the output of the first function (`func1`), while Comm_1 only raises an alert about its input, as it is unable to properly process the function (because of the missing red edge in the first function, cf. Figure 1). On the other hand, Comm_2 reports only the input for the second function (`func2`), because of the missing green edge in its model (cf. Figure 1), while Comm_1 does not raise any alert, as its analysis is still blocked by the first function.

In other words, the interplay between different tarpits that affect different tools result in the fact that none are able to process the entire chain during the first iteration. Therefore, our infer and stitch operations need to be repeated in an iterative fashion until an equilibrium is reached, i.e., until no new edges (new stitches) are discovered in the graph. To sum up, in the first iteration

our approach helps Comm_1 to understand that `$x` is tainted, and thanks to this information during the second iteration Comm_1 detects that `$y` is tainted as well – a piece of information that helps Comm_2, which previously missed this connection.

Figure 2 shows the data-flow graph of the modified application after two iterations of our approach. The new edges, marked in blue, are the stitches introduced by our approach. In the third iteration no new stitches are inferred and our approach moves to the evaluation phase.

4.3 Phase III: Evaluate

In this final phase (cf. lines 19-22 of Algorithm 1), our approach first cleans the application code from any remaining fake sinks (cf. `clean` instruction at line 20) and then scans it with all SAST tools in the arsenal. By removing from these SAST findings for vulnerability class V those already reported on the original application (cf. `diff` at line 22), our approach can output the novel SAST findings emerging because of the stitches added in the previous phase. In order to remove already reported SAST findings, our `diff` instruction compares findings as follows. Two findings $F1$ and $F2$ are considered identical if and only if the sink line of $F1$ is identical to the sink line of $F2$.

For our motivational example, the comparison is trivial as there were no findings reported on the original application. By running our infer and stitch operations, new stitches were added in the first two iterations and none were uncovered during the third iteration. When scanning the final code, both SAST tools in our arsenal were correctly reporting the XSS vulnerability, indicating that two new findings emerged as a direct consequence of our approach that forced the SAST tools to collaborate.

5 False Positives and False Negatives

SAST tools employ various techniques to analyze the source code of an application. In particular, to handle dynamic code constructs that cannot be resolved statically, all SAST tools use some form of over- and under-approximation. The two are often combined to find a balance between the amount of code that can be analyzed and the number of false alerts generated [5]. Additionally, SAST tools also incorporate heuristics to halt the analysis of a particular path when specific thresholds are reached (e.g. if the path involves over 500 variables or more than 5 nested functions). The aim is to improve performance and keep the running time in a reasonable range, but this approach can decrease the ability of the tool to detect vulnerabilities.

Our approach reduces false negatives by allowing all tools to access data-flow connections that emerge by combining and complementing the models of each individual tool. As an additional benefit, our approach also reduces the impact of performance-related thresholds since, by adding stitches, we introduce shortcuts that bypass functions and make data-flow paths shorter. This again helps in reducing false negatives. It is also important to note that, by construction, our approach cannot increase false negatives as it cannot miss what individual tools would already detect in isolation. This is also confirmed experimentally in our results, where we never encountered a target application for which our approach was not detecting a finding that was previously detected by one of the SAST tools.

In the majority of cases (around 80% of the applications analyzed in our experiments), our approach is returning novel SAST findings. Although the majority of these findings are false positives, which is a common occurrence with most SAST alerts, they all result from the examination of novel security-related data-flow paths. In fact, our approach only requires SAST tools to analyze a path if it can be decomposed into a sequence of sub-paths, each of which was already analyzed by at least one SAST tool because the tool believed it may transfer unsanitized user input. Therefore, our algorithm does not introduce more false positives *unless* one of the tools already introduced them because of an over-approximation. In this case, the infer-and-stitch approach enforces the same over-approximation on the other tools, causing all of them to consider the corresponding, potentially erroneous, path.

6 Experiments: methodology

We implemented our approach in a prototype tool named WHIP. WHIP takes as input a web application and one or more types of sinks (which depend on the class of vulnerabilities the analyst wants to discover) and then orchestrates the insertion of sinks, the execution of SAST tools and the collection of the corresponding alerts, and the injection of conditional assignments instruction to create the new edges in the dataflow graph. The tool automatically performs multiple iterations until the data-flow graph converges and no new edges are inserted.

WHIP currently supports the PHP language, chosen because it is still by far the most common language to develop Web applications, with a 78% market share in 2022[1]. On the other hand, since we are building our

stitches at the function call level and WHIP does not require any static analysis, our approach can be applied to any programming language supported by SAST tools. To support a new language, the analyst just needs to list the sinks statements used by the language and the syntax required to assign variables. In the rest of this section, we discuss the integration of SAST tools into WHIP, the selection of tools we used in our experiments, and the dataset of Web applications we tested with WHIP. The results obtained are presented in Section 7.

6.1 SAST Tools Integration

Since WHIP needs to orchestrate the execution of SAST tools, it requires a dedicated module to support the interaction with each tool and the parsing of the generated alerts. So the integration of a SAST tool within WHIP requires the implementation of a small interface that handles its operation and the collection and inspection of its alerts.

In order to add a static tool to WHIP, two conditions must be met: (1) the static tool needs to have a CLI interface or some form of API to control its operation, and (2) the output (alerts) of the tool need to be stored in a way that allows for an automated extraction and parsing. These requirements come from the fact that WHIP is fully automated and requires the ability to orchestrate the process. For instance, if the tools offer CLI commands to scan a project and produce results, to use them with WHIP, we run the scan command and redirect their output to a text file. Then, we wrote a script to parse the text file and extract the data required by WHIP. The simple parsers we developed for some tools are available in our repository [4].

It is typical for commercial SAST tools to provide in their reports a list of findings along with the *type* and *severity* of each entry. The severity was not important in our experiments, since WHIP focuses on injection vulnerabilities [2], which are considered by all SAST tools as high severity risks. In addition, for each alert, SAST tools may also report the full path between the corresponding source and sink (i.e., the attacker-controlled input and the point in which the vulnerability is located in the code) to help developers to understand the issue and provide a patch.

6.2 SAST Tools Selection

While our solution is generic and it can be applied to any SAST tool, it is particularly useful in the presence of commercial, closed-source tools whose source code cannot be modified to take into account other complementary solutions. For this reason, for our experiments we selected two of the most widely used commercial SAST tools that support PHP: Comm_2 and Comm_1. We have acquired full licenses for both tools, allowing us to run our tests without the restriction and limited analysis time available in the free trial options.

On top of them, we also decided to include in our tests a few selected open source (OS, in short) tools to verify whether their presence could benefit the results by helping commercial tools to overcome their limitations. In fact, while much more limited in terms of complexity and supported features, it would be enough for an OS tool to discover a data-flow relationship missed by the more mature commercial alternatives to provide a valuable contribution to the overall *collaborative effort*.

Over the years, the research community has proposed several static vulnerability detection tools for PHP to choose from (including RIPS [12], phpSAFE [30], WAP [38], Progpilot [32], WeVerca [14] and Pixy [18]). For our experiments we selected Progpilot v1.0.2 and WAP v2.1 because they both support scanning entire projects instead of individual files, they both provide a CLI implementation, and they both support object-oriented code. In addition, Al Kassar et al. [6] found Progpilot to be the best OS tool in terms of its ability to handle the authors' testability tarpits library.

6.3 Dataset

To measure the benefits of combining multiple SAST tools, we tested our tool (WHIP) on a set of modern and popular web applications. We cloned the latest version of all PHP projects from Github with more than 1,000 stars, resulting in an initial set of 1,183 projects. We then extracted the number of sources of user-provided inputs in each project, by grepping for the corresponding predefined variables in PHP (e.g., $_GET and $_POST). Roughly half of the projects (602 projects) do not have any source. This is due to the fact that these are often libraries used by other projects and not standalone applications. Among the remaining projects, 127 contained more than 100 sources – thus making them a perfect target for our vulnerability analysis. Thus, we selected these 127 applications as a dataset for our experiments.

The analysis was performed on a machine with 16 cores and 64 GB of RAM. Since each SAST tool needed to be invoked multiple times for each project, we excluded those for which a single analysis did not complete within 6 hours. This was the case for 13 projects, reducing our final dataset to 114 projects. The complete list of applications with their corresponding statistics is reported

in Appendix in Table 4. The projects range from small (with less than 10K LoC) to big (with more than 1M LoC). Altogether, they account for 21.4M LoC, 1.9 million functions, and 85K sources of user-provided input.

7 Experiments: Results

We break down the results of our experiments in six different parts. First, we will examine the poor performance and lack of contribution of open source tools. Then, by using XSS as an example, we will analyze different statistics that show how WHIP performed on the 114 projects in our dataset, including the number of iterations required to converge and the number of additional edges that WHIP introduced in the PHP code.

In the third part, we will evaluate the impact of WHIP by measuring the number of extra alerts raised by each SAST tool for three types of injection vulnerabilities (XSS, SQLI, and fileM). An increase in the number of alerts indicates that the SAST tools were able to analyze the application code more deeply and process more paths that were previously blocked by the presence of testability tarpits. In the fourth part, we will discuss the advantages and disadvantages for companies that will use WHIP. On the one hand, the tool will increase the coverage of the source code and de-duplicate the alerts from multiple tools. On the other hand, it will require more time and resources to scan the project using multiple SAST tools over a few iterations.

In the fifth part of this section, we show that the model of the application built by SAST tools thanks to WHIP is *more* than the sum of the individual models. In other words, it is not just that one tool can help the other to overcome its limitations, but that each tool can now discover vulnerabilities that could not be previously discovered by any tool in isolation.

Last, we compared the vulnerabilities we discovered with a dataset of 100 past CVE reports in Appendix B. By analyzing the length and complexity of the data-flow paths associated with each bug, we found that our discoveries have longer paths than the average reported vulnerabilities. The average number of lines of code in the past CVE reports was 7.8, while in our discoveries, it ranged from 12 to 53 LOC, with an average of 24.9.

7.1 Research Tools

In our experiments, Progpilot produced results only for 26 out of 114 projects (23%) and crashed in the remaining cases. WAP did better, successfully scanning 90 projects (79%). In both cases, the research tools did not report any additional alerts (i.e., no new edges) on top of those reported by commercial tools, leading to the conclusion that these tools could not be used to enhance the analysis of commercial tools. This poor result is not completely unexpected. In fact, in 2017 Nunes et al.[28] already noticed that none of the static research tools for PHP (RIPS, phpSAFE, WAP, Pixy, and WeVerca) were able to successfully analyze a complex web application in its entirety. Similarly, Al Kassar et al.[6] found that only commercial SAST tools were up-to-date with recent PHP language features and capable of scanning modern web applications and that the two leading commercial tools supersedes all other open source alternatives in terms of supported testability tarpits. For this reason, we focus in our main experiment on commercial tools and include an experiment about research tools in Appendix A.

In conclusion, adding small tools to the arsenal has a cost (in terms of analysis time) but may not bring any clear benefits to security testing. On the other hand, in the next sections we will see how the combination of state of the art solutions can instead lead to a large increase both in terms of code coverage and number of alerts and discovered vulnerabilities.

Iteration	0	1	2	3	4	5	6	Total
Projects	9	31	33	23	10	3	5	114

Table 1: Number of Converged projects over iterations

Iteration	1	2	3	4	5	6	Total
Comm_1	5,475	1,847	365	131	68	46	7,932
Comm_2	6,976	1,450	462	240	144	104	9,376
Combined	12,451	3,297	827	371	212	150	17,308

Table 2: Inserted data-flow edges with XSS fake sinks

7.2 General Statistics

Table 1 shows the number of iterations required by WHIP to converge (i.e., until no more data-flow edges were discovered) for different projects when we run the experiment for XSS vulnerabilities. At each iteration, each SAST tool was able to explore the application deeper, i.e., to test the security of longer and longer inter-procedural data-flow paths that were invisible (or better, fragmented) without our approach. One iteration was sufficient to converge for 31 of our 114 applications, two could cover roughly half of the dataset, and the rest required three or more iterations – with a maximum of six. This is very important, as it means that some applications contained paths between a source and a sink that involved at least six different functions, all of which contained snippets of code that both our tools were not able to process correctly (different tools had problems

with different code blocks in alternation).

Table 2 shows the number of edges added by WHIP to the applications code, for each iteration and each SAST tool. By far, the largest number of edges (12.4K and 3.2K respectively) were added over the first two passes. In total, our approach added 17,308 new edges: 7.9K to increase the coverage of Comm_1 and 9.4K to increase the coverage of Comm_2.

It is also interesting to observe that not all applications were impacted the same way. For instance, the Dolibarr application contained the maximum number of sources in our dataset (8,609 sources) with more than 8 million lines of code. This project was also the one for which our tool had the largest effect, introducing 108 new stitches for Comm_2 and 206 for Comm_1 after four iterations. At the other end of the spectrum we find the Valet-plus project, which has the minimum number of sources in our dataset (100), and for which our tool only added two stitches for Comm_2 in the first iteration. Overall, WHIP added 151.82 stitches per project, with a median of 35. More details are presented in Table 4 in Appendix, where for every project we show the number of stitches added and the number of iterations run by WHIP.

7.3 New Alerts

To test for new alerts we run three separate experiments, using WHIP to test high severity injection vulnerabilities such as XSS, SQLi, and file manipulation (fileM) vulnerabilities. As a reminder, the stitches we introduce are always specific for a class of sinks as mixing two types can lead to errors in the alerts. For instance, if a function propagates information unsanitized for XSS but sanitized for SQLi, adding a stitch would result in a 'shortcut' that can make tools erroneously report SQLi vulnerabilities (since the alternative path we insert would bypass the sanitization). Therefore, when we add stitches for a given type of sink, we need to later test only for vulnerabilities that involve the same sink type.

For the three experiments we collected all the corresponding alerts (XSS, SQLi, and fileM, depending on the experiment) raised by the SAST tools when scanning (i) the original version of each application as well as (ii) the modified versions generated by WHIP.

Overall, Comm_2 went from 49,231 alerts on the original applications to 61,583 (+25.1%) on the stitched versions. These were divided in 52,843 (versus 42,040) XSS, 1,789 (vs 1,744) SQLi, and 6,861 (vs 5,447) fileM. Comm_1 alerts increased instead from 50,217 to 54,985 (+9.49%), divided into 33,028 XSS (vs 30,062), 10,284 SQLi (vs 9,236), and 11,673 fileM (vs 10,919).

The complete breakdown of the discoveries for each project is presented in Table 4. Even if we expect the majority of these alerts to be false alarms (as we will discuss in more details in Section 7.5), these numbers show that both SAST tools were able to improve the number of potentially vulnerable data flows by roughly 25% for Comm_2 and 10% for Comm_1.

We can further divide these sets of new alerts into two different categories: Known and Unknown. The first contains new alerts that are generated by one tool (with the help of the other), but that the other tool was already able to discover by itself. These alerts are associated with source-sink paths in the data-flow graph that only contain new edges for one tool. The second category (Unknown alerts) contains instead the alert that one tool generated (with the help of the other) but that none of the tools were able to discover alone. In this case, the path associated with the alert passes through stitches for both tools.

Figure 7.3 shows the breakdown of the Known and Unknown alerts for the two tools and for the three vulnerability types. We can see that, over the 114 projects, Comm_1 was missing around 1,936 known alerts (1530, 151, and 255 for XSS, SQLI and fileM), which were already reported by Comm_2. On the other hand, Comm_2 was missing 5,864 known alerts (5076, 84, and 704 for XSS, SQLI and fileM) that were detected by Comm_1. But it is much more important to focus on the curves of the alerts that were previously unknown. In this case, we can notice that a stunning 9,226 alerts with the highest priority (respectively (i) 2,742 reported by Comm_1: 1436 XSS, 897 SQLI, and 409 fileM and (ii) 6,484 by Comm_2: 5727 XSS, 51 SQLI, and 706 fileM) were raised for the first time thanks to our tool.

If we look at the different applications in our dataset, 65 out of 114 show an increasing number of high severity alerts for Comm_2, with an average number of new alerts of 108.35. Comm_1 reported instead new alerts in 61 projects, with an average of 41.82. At a closer look, we can observe a clear relationship between the number of sources, the number of stitches, and the number of alerts. For instance, `Dolibarr`, `phpipam` and `testlink-code`, the three projects with the highest number of new alerts, all have more than 900 sources each, and required more than three iterations for WHIP to converge. On the other hand, out of the 35 projects that have less than 200 sources, 19 did not have any new discovery by Comm_2 and 21 had zero new alerts for Comm_1. This seems to suggest again that the more complex the application is, the more likely it is to benefit from our approach, and the higher is the number of new alerts generated by SAST tools.

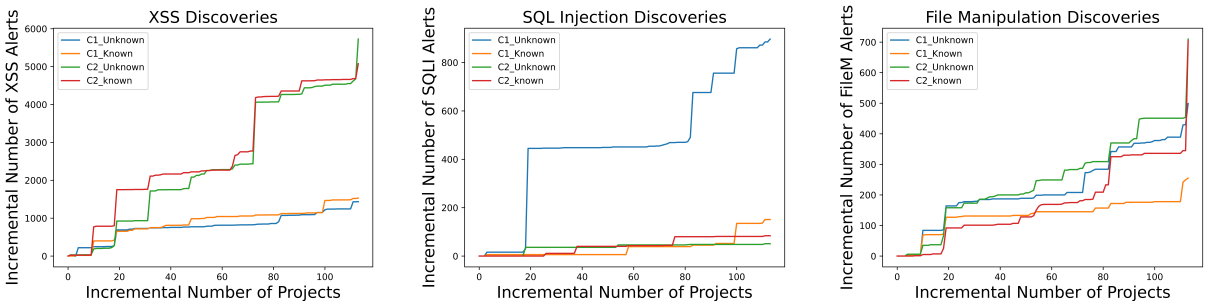

Figure 3: The incremental number of alerts regarding the number of projects

7.4 Overhead

Our approach offers many benefits for enterprises. First of all, our solution increases the coverage of the source code and allows existing tools to report new alerts and detect new vulnerabilities. In addition, WHIP provides a solution for companies that have invested in multiple SAST tools but are facing difficulties in effectively integrating them. Our approach enables the optimization of existing tools and can also save time for testers. In fact, in our experiments we identified 44,000 duplicate alerts that were separately reported by Comm_1 and Comm_2, which our tool automatically de-duplicates and reports only once.

The downside of our approach is the increased scanning time and resource consumption, as static tools must be re-executed over multiple iterations. In our experiment, running Comm_2 alone for one iteration on all projects took 11 hours, 23 minutes, and 21 seconds (6 minutes per project on average). Comm_1 required 36 hours, 6 minutes, and 57 seconds (with an average time of 19 minutes per project). Due to the additional iterations, WHIP took a total of 164 hours, 36 minutes, and 11 seconds to run all experiments until all projects converged. Thus, on average WHIP required 86 minutes per project, versus 25 minutes required by the two commercial tools in isolation (corresponding to a slowdown factor of 3.4X).

A company can also decide to limit our approach to three iterations (reducing the slowdown to 2.6X), a good compromise if we think that only 18/144 projects converged after three iterations, and that 83% of the new alerts were reported over the first three iterations.

7.5 New Discoveries

As we previously explained, the use of our technique allowed the two tools to report 9,226 completely new high severity alerts, discovered after adding at least two stitches. To conclude this section, we now look at those alerts and, through a process of manual validation, try to separate false positives from zero-days vulnerabilities.

Since the number is too high for a complete and thorough manual investigation, we started by randomly sampling 10 alerts for each project, for a total of 640 alerts. If at least one of the ten alerts was confirmed as true positive, we proceeded to verify all the other alerts for the same project, to check how many alerts will be fixed when we fix this real vulnerability. In total, this resulted in a set of 2,732 alerts we manually investigated. Each time we confirmed that an alert was a true positive, we contacted the developers to initiate a process of responsible disclosure. In each communication we described the issue and provided feedback on how the vulnerability could be fixed, and in some cases even submitted ourselves pull requests on Github containing the patch.

At the time of submission, we identified 35 zero-day vulnerabilities in 14 projects. Developers have confirmed **24** of these vulnerabilities (from 9 different projects), as shown in Table 3. The remaining 2697 alerts we investigated turned out to be false alarms. The fact that 98% of the validated alerts were false positives should not be a surprise as SAST tools are known, unfortunately, for their very high false positive rates. With our approach, the total number of generated alerts increased by roughly 10%. In fact, without WHIP, Comm_2 and Comm_1 already reported a stunning 99,448 alerts. If we consider the fact that we only scanned very popular projects that are regularly analyzed with SAST tools for security purposes, we can expect the very vast majority (if not all) of these alerts to be false alarms.

Table 3 reports the number of vulnerabilities aggregated in groups, based on how they were handled by the developers. For instance, in the Phoronix application (line 7 in the table) our system found several ways to bypass the sanitize function the authors used in their project. While each way is an independent discovery and therefore a true positive alert, the developers were able to fix all of them by changing the sanitizer, and thus we only requested one CVE covering all the corresponding cases. On top of those already accepted, we also reported

	Type	Project	Stars	Discoverd_By	Vuls	Status	CVE	FUNC	LEN	Stitch
1	XSS	Vesta	2700	Comm_1	1	Confirmed	CVE-2022-36305	6	23	2
2	XSS	Jukebox-RFID	1000	Comm_2	4	Confirmed	CVE-2022-36749	2	12	2
3	XSS	Cacti	1200	Comm_2	3	Confirmed	Requested	8	33	2
4	File M	ICEcoder	1400	Comm_2	2	Confirmed	CVE-2022-34026	4	14	2
5	XSS	Dokuwiki	3500	Comm_1	1	Confirmed	CVE-2022-28919	4	13	2
6	XSS	PicUploader	1000	Comm_1	4	Confirmed	CVE-2022-41442	4	16	2
7	XSS	Phoronix	1700	Both*	7	Confirmed	CVE-2022-40704	14	43	2
8	XSS	Librenms	2800	Comm_2	1	Confirmed	CVE-2022-36746	7	32	2
9	XSS	Phpipam	1700	Comm_2	1	Pending	CVE-2022-41443	5	17	2
10	File M	Dzzoffice	3500	Comm_2	7	Pending	-	4	11	2
11	XSS	Razor	1100	Comm_2	1	Pending	CVE-2022-36747	7	18	2
12	XSS	Pfsense	3900	Comm_2	1	Confirmed	CVE-2022-42247	4	16	3
13	XSS	Carbon-Forum	1800	Comm_2	1	Pending	-	8	47	4
14	XSS	SuiteCRM	3100	Comm_2	1	Pending	-	11	53	4
	SUM	14			35					

Both*: In Phoronix project, five discoveries deteced by Comm_2 and two detected by Comm_1

Table 3: New Vulnerabilities Detected with Our Approach

11 other potential vulnerabilities in five projects, for which we did not yet receive an acknowledgment from the developers. Table 3 also shows the number of stitches required to discover each vulnerability. Among our findings, 32 vulnerabilities were discovered with two stitches, one with 3 stitches, and 2 after the insertion of 4 stitches.

At first, one might think that missing data-flow edges (the main contribution WHIP helps to mitigate) is only a tiny factor among many other limitations that affect today's SAST tools. However, it is important to stress that missing edges is *NOT* a limitation per se, but only the consequence of a multitude of other actual limitations. In other words, many problems – from the inability to support certain language features, to missing models of API functions, to the inability to correctly reconstruct inter-procedural control flows, to under-approximation in resolving dynamic behaviors – ultimately result in the inability of a tool to detect the data-flow link among two parts of a program. Given the nature of injection vulnerabilities, these missing edges (independently from the reason why they are missing) are the main cause of undiscovered vulnerabilities in complex real-world applications.

Ethical Risk Assessment. In this study, we responsibly disclosed 35 zero-day vulnerabilities that we detected using our approach. We validated these vulnerabilities by manually checking 2,732 out of 9,226 alerts generated by WHIP. Due to the large number of alerts, we were unable to check them all. If the ratio remains the same, we could expect another 83 vulnerabilities to be present in the remaining 6,494 unvalidated alerts. The names of the static tools used to detect the alerts will be kept anonymous, and we will not publish any information about the non-reported alerts. Finally, we promise to delete all alerts from our servers after the paper has been reviewed and accepted.

8 Related work

We can identify three research directions related to our work. First, researchers have conducted extensive tests of different SAST tools and they concluded that none of them outperforms the other in all situations. Second, researchers have tried to mitigate the shortcomings of a single tool by either 1) combining static and dynamic techniques, 2) resorting to human experts to help the tool perform better, or 3) combining multiple tools and joining the results. However, our approach is the first that, by combining the internal knowledge of different tools, allows a set of tools to discover more than the sum of the individual components.

Tools Comparison. Many research papers analyze static tools to demonstrate that there is no tool that supersedes all others. Nunes et al. [29] introduce a benchmark for comparing static analysis tools and their effectiveness in detecting security vulnerabilities. The authors chose five static tools for PHP and they found that the best tool varies from one scenario to another, depending on the vulnerability class. Algaith et al. [7] compare the same five static tools as well as their combinations. The authors pointed out that none of the tools (or combinations thereof) can discover all the vulnerabilities in their dataset, but a set of three gave the maximum number of discoveries.

While most of the research papers compare open-source tools, few papers also include commercial tools. For instance, Al Kassar et al. [6] include three tools,

Spoto et al. [35] six and Kupsch et al. [19] compare Fortify and Coverity on the analysis of a single project.

Tools Collaboration. In this work, we present a new direction to enable static tools to collaborate and discover more vulnerabilities. To the best of our knowledge, there are no other works in this area, previous studies have looked at other forms of collaborations – either between static and dynamic tools, or between static tools and human developers.

[I] Static + Dynamic. Static tools have obvious limitations when it comes to handling dynamic features [15, 16, 20], such as indirect function calls. NAVEX [8] tries to overcome these limitations by proposing the collaboration between static and dynamic approaches. In this paper, the authors proposed to use a crawler to capture the relationship between different web pages, and then use this information to complement a static analysis performed on the application's source code. We can distinguish two cases. If the name of the file is defined statically, then there is no need for the dynamic approach because SAST tools can already detect this file. If, on the other hand, the file is included dynamically then none of the static tools can detect this file, and therefore the solution presented in this paper does not help. Thus, we believe the two techniques to be orthogonal with no intersections in their findings.

Other studies used dynamic techniques to *verify* the discoveries of static tools. For example, Csallner et al. [11] used a hybrid analysis approach to automate bug findings. The proposed approach includes three steps: dynamic inference, static analysis, and dynamic verification. The same authors [10] also presented a different approach in which a constraint solver was used to generate concrete test cases to verify the static tools alerts. There are also other types of collaborations, like the one proposed by Hough et al. [17], in which the authors employ human developers' test suites to support automated dynamic analysis.

[II] Static + Humans. A different form of collaboration that has been explored by researchers to overcome some of the limitations of SAST tools is based on human-in-the-loop approaches. For instance, Al Kassar et al. [6] discuss the collaboration between SAST tools and developers when they provide manual transformation at the source code level to improve the discoveries of the static tools. Other authors study how to change the rules automatically depending on the users' preferences (e.g., in Mangal et al. [24]), or how to provide feedback to the tool's developers to improve the results (e.g., in Sadowski et al. [34]).

Combining the results of multiple tools. Many papers have proposed to combine the output of different static tools, as suggested by NIST "CAS Static Analysis Tool Study Methodology" [25]. Meng et al. [26] show that static tools for JAVA can report different alerts for the same source code depending on the tool performance in that specific class of bugs. So they ask the analyst to provide the source code and an example of the set of bugs she is looking for. The system then chooses the right tools that are most likely to provide the best results for that type of bugs and returns a merged list of their discoveries. Rutar et al. [33] suggest a bug-finding meta-tool for joining the results of different static tools together. Wang et al. [37] introduce a web service where the user can choose the type of bugs and upload the source code. The system then scans the code with multiple tools and returns the merged results to the user. Finally, Nunes et al. [28] run an empirical experiment on combining the results of five static tools for web applications, reporting the increased percentage of the true-positive and false-positive after this combination.

9 Conclusion

In this paper, we proposed a novel idea to 'force' different SAST tools to collaborate to find more vulnerabilities. Whereas each static tool has its own strengths and weaknesses, our solution allows them to help each other to overcome their respective challenges. Our approach is completely automated and considers all tools as black boxes (thus supporting commercial tools for which we have no visibility on their internal data structures).

By routinely modifying the source code of the application under test, our system can inject fake sinks to infer how tainted values propagate through the different program functions. Whenever one of the tools is unable to 'understand' these connections, we help it by stitching the data-flow with additional edges that bypass the problematic function.

Our experiments performed on a large set of popular PHP applications, show that our approach can successfully improve the amount of source-to-sink paths that each tool is able to analyze. This leads to a total increase in the number of critical alerts between 10-12%. By manually validating a subset of these new alerts, we discovered and reported 35 zero-day vulnerabilities in 14 projects with more than 1,000 stars on Github.

Acknowledgment

This work has received funding from the European Union's Horizon 2020 research and innovation programme under project TESTABLE, grant agreement No 101019206.

References

[1] Comparison of the usage statistics of PHP vs. ASP.NET for websites. https://w3techs.com/technologies/comparison/pl-aspnet,pl-php.

[2] OWASP top ten security risks - Injection. https://owasp.org/Top10/A03_2021-Injection.

[3] Vulncode-DB. The vulnerable code database. https://www.vulncode-db.com.

[4] WHIP Repository. https://github.com/enferas/WHIP.

[5] Filtering false alarms of buffer overflow analysis using smt solvers. *Information and Software Technology*, 52(2):210–219, 2010.

[6] Feras Al Kassar, Giuliad Clerici, Luca Compagna, Fabian Yamaguchi, and Davide Balzarotti. Testability tarpits: the impact of code patterns on the security testing of web applications. NDSS 2022, 2022.

[7] Areej Algaith, Paulo Nunes, Fonseca Jose, Ilir Gashi, and Marco Vieira. Finding sql injection and cross site scripting vulnerabilities with diverse static analysis tools. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 57–64, 2018.

[8] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, Baltimore, MD, August 2018. USENIX Association.

[9] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 334–349, 2017.

[10] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 422–431, New York, NY, USA, 2005. Association for Computing Machinery.

[11] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), may 2008.

[12] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. 01 2014.

[13] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static analysis of java dynamic proxies. ISSTA 2018, page 209–220, New York, NY, USA, 2018. Association for Computing Machinery.

[14] David Hauzar and Jan Kofron. Framework for Static Analysis of PHP Applications. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 689–711, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] M. Hills. Variable feature usage patterns in php (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 563–573, 2015.

[16] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of php feature usage: A static analysis perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 325–335, New York, NY, USA, 2013. Association for Computing Machinery.

[17] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. Revealing injection vulnerabilities by leveraging existing tests. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 284–296, 2020.

[18] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy*, pages 6 pp.–263, 2006.

[19] James A. Kupsch and Barton P. Miller. Manual vs. automated vulnerability assessment: A case study. In *In First International Workshop on Managing Insider Security Threats (MIST), West*, 2009.

[20] Panos Kyriakakis, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, and Stelios Xinogalos. Exploring the frequency and change proneness of dynamic feature pattern instances in php applications. *Science of Computer Programming*, 171:1–20, 2019.

[21] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of java reflection - literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518, 2017.

[22] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 544–555, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis, 2005.

[24] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. ESEC/FSE 2015, page 462–473, New York, NY, USA, 2015. Association for Computing Machinery.

[25] George Gordon Meade. Cas static analysis tool study - methodology. 2013.

[26] Na Meng, Qianxiang Wang, Qian Wu, and Hong Mei. An approach to merge results of multiple static analysis tools (short paper). In *2008 The Eighth International Conference on Quality Software*, pages 169–174, 2008.

[27] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166, 2016.

[28] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. On combining diverse static analysis tools for web security: An empirical study. In *2017 13th European Dependable Computing Conference (EDCC)*, pages 121–128, 2017.

[29] Paulo Nunes, Ibéria Medeiros, José C. Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018.

[30] Paulo Jorge Costa Nunes, José Fonseca, and Marco Vieira. phpsafe: A security analysis tool for oop web application plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 299–306, 2015.

[31] OWASP. Code Review Guide v2. https://owasp.org/www-project-code-review-guide/assets/OWASP_Code_Review_Guide_v2.pdf.

[32] progpilot. progpilot - A static analyzer for security purposes. https://github.com/designsecurity/progpilot.

[33] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, page 245–256, USA, 2004. IEEE Computer Society.

[34] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608, 2015.

[35] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. Static identification of injection attacks in java. *ACM Trans. Program. Lang. Syst.*, 41(3), July 2019.

[36] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation, 09 2018.

[37] Qianxiang Wang, Na Meng, Zhiyi Zhou, Jinhui Li, and Hong Mei. Towards soa-based code defect analysis. In *2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 269–274, 2008.

[38] OWASP WAP. Web Application Protection Project. https://securityonline.info/owasp-wap-web-application-protection-project.

[39] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with

code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

A Experiment on Open Source Tools

The technique presented in this paper is designed to leverage the *strengths* of multiple SAST tools. Therefore, including a tool that does not provide any benefit only results in an increase in the overhead. As previous studies have shown, open source static analysis tools for PHP are unable to cope with today's code complexity, they often lack support for object-oriented code, and are far behind their commercial counterparts in terms of discoveries and false alarms. Indeed, as we discussed in the paper, including open source tools in our suite did not lead to any new data-flow paths nor new findings on top of those reported by commercial solutions.

However, if a user has only access to open source solutions, our approach can still be beneficial, by allowing a simple integration of different techniques without requiring to merge complex code bases of existing tools. To test this hypothesis, we repeated a second time the experiment described in Section 6, this time using only open source tools: Progpilot and WAP.

As discussed in Section 6.1, to integrate a new tool in WHIP, a small interface needs to be implemented to handle its operation and to collect and inspect its alerts. Progpilot and WAP provide CLI commands to scan projects and generate results. To use these tools, we execute the scan command and redirect the output to a text file. We then created a script to parse the text file and extract the data needed by WHIP. The parser we developed is available in our public repository [4]. The repository also includes an example that illustrates a snippet of code containing a vulnerability with two testability tarpits in the path between the source and the sink. The first tarpit (the use of certain arithmetic operations on strings) is handled correctly by WAP but not supported by Progpilot, while the second (a wrong sanitizer) does not cause problems in Progpilot but is not recognized by WAP. Running WHIP on this example shows how the two tools can 'collaborate' to jointly analyze the code, resulting in the fact that both tools are able to detect the vulnerability after two iterations.

In our dataset, only 26 out of 127 projects where successfully analyzed by both Progpilot and WAP. In all other cases, one of the two tools (or both) crashed during the analysis. In addition, these projects were simpler and smaller, accounting for only 7% of the total lines of PHP code in our dataset. Out of the 26 projects, 5 converged after 3 iterations, while the rest converged after two. The number of alerts reported by WAP increased by 7%, while the number of alerts of Progpilot increased by 15%. Thanks to WHIP, the two tools were able to report 194 previously unknown alerts, which were not discovered by either one or the other in isolation. Even though, as expected, all the reported alerts were already reported by our commercial solutions, this still shows that our technique can help improve the results of any combination of tools.

B Vulnerabilities Complexity

Our approach not only helps tools to discover more vulnerabilities, but also to discover vulnerabilities associated with long data-flow paths, which can be difficult for analysts to discover even through manual inspection. To support this hypothesis we built a dataset of 100 vulnerabilities (CVEs) from Vuln-code DB [3], for which there was enough information to reconstruct the vulnerable source-to-sink path. This requires the corresponding patch to clearly distinguish between the security fix and other changes in the source code, and the CVE to contain an example of input to reproduce the bug. By using this information we manually reconstructed, for each vulnerability, the path between the source and the sink through a manual inspection of the source code. Over the 100 vulnerabilities in our dataset, the average number of functions traversed by these paths is 3.4 (with a median of 3), while the average number of lines of code is 7.8 (with a median of 5).

If we compare these values with the ones associated with the vulnerabilities discovered by WHIP (as reported in Table 3) we can notice a clear difference. For instance, the length of the paths among the new vulnerabilities we discovered span from 12 to 53 lines of code (with a mean of 24.9 and a median of 17.5). Thus, if we take the length of the vulnerable path as a possible measure of the complexity of a vulnerability, our approach results in vulnerabilities that are, on average, three times more complex than those regularly reported by other means.

This is due to the fact that our solution helps SAST tools to overcome their limitations and therefore to explore deeper in the applications code and detect vulnerabilities associated with long inter-procedural paths.

C Information and Results

Table 4: Dataset: Information and Results

ID	Project	Project info				Comm_2			Comm_1			Stitches	Iter
		stars	functions	LoC	sources	before	after	diff	before	after	diff		
1	easyappointments	2281	4561	54619	139	0	0	0	19	19	0	9	2
2	ampache	3074	9894	127334	662	12	66	54	557	557	0	29	1
3	amp-wp	1750	8038	111224	213	0	0	0	1	1	0	38	1
4	sqli-labs	4092	107	6791	214	77	77	0	83	122	39	63	3
5	jetpack	1402	20737	238372	1080	25	32	7	139	365	226	248	3
6	bjyadmin	1743	29217	178780	397	491	492	1	40	40	0	39	2
7	CodeIgniter	18183	3657	35663	308	0	0	0	18	18	0	6	2
8	boinc	1500	22237	124889	385	18	20	2	240	243	3	20	2
9	brefphp	2512	965	6867	145	7	7	0	5	5	0	0	0
10	upload-labs	2872	54	2649	129	25	25	0	43	43	0	11	3
11	cacti	1242	19825	169391	402	5429	6364	935	2397	2948	551	539	3
12	cashmusic	1166	60516	501394	1028	125	153	28	289	294	5	163	4
13	organizr	3914	56312	161912	293	10	10	0	46	46	0	1	1
14	unmark	1544	3151	40325	167	4	6	2	29	29	0	12	2
15	razor	1127	12572	91366	354	55	63	8	431	431	0	66	1
16	CodeIgniter4	4205	9602	110251	776	8	8	0	10	14	4	9	1
17	DVWA	6335	1555	21169	217	28	28	0	33	33	0	18	2
18	diskoverdata	1180	3598	34282	163	661	678	17	277	277	0	29	2
19	docker-labs	10842	23412	167147	1576	1444	1591	147	138	197	59	668	4
20	dolibarr	3339	55001	897702	8609	17249	19010	1761	13846	15044	1198	3860	6
21	drupal	3735	60469	834417	208	0	0	0	41	41	0	7	1
22	elementor	4519	8661	69504	102	4	4	0	20	20	0	18	4
23	facebook-php	3303	238	2794	115	5	5	0	2	2	0	0	0
24	yii2-fecshop	4952	17751	174633	118	4	6	2	162	162	0	11	2
25	loklak-wp	1511	28212	437612	1980	2191	2191	0	262	322	60	780	3
26	phimpme-drupal	1537	74385	699289	618	0	0	0	39	40	1	47	3
27	phimpme-wp	1534	9824	114294	1397	1224	1272	48	155	215	60	554	3
28	freescout	1563	36940	304065	229	0	0	0	56	56	0	14	1
29	FreshRSS	4542	2597	65362	168	29	29	0	82	82	0	3	1
30	Froxlor	1441	1395	60567	453	10	10	0	172	172	0	59	2
31	rhaphp	1078	15980	58617	201	5	9	4	54	55	1	15	2
32	Geocoder	3814	1453	17788	240	0	0	0	35	35	0	0	0
33	glpi	2691	13103	349606	3719	780	1925	1145	1223	1278	55	626	2
34	imgurl	1566	2542	36465	164	1	2	1	56	57	1	20	2
35	scws	1573	417	6722	121	12	36	24	105	105	0	6	1
36	VueThink	1347	1998	24799	119	19	53	34	50	50	0	21	6
37	icecoder	1352	1515	11963	228	192	207	15	137	137	0	21	1
38	php-webshells	1713	1114	67034	1986	2519	2541	22	2401	2435	34	133	2
39	openflights	1187	3189	10642	133	331	362	31	117	152	35	69	3
40	KodExplorer	5685	5197	58321	187	76	85	9	31	33	2	18	2
41	php-benchmark	1032	12335	155976	886	3	3	0	102	107	5	40	3
42	laravel	27494	23460	186354	443	0	0	0	2	2	0	1	1
43	Leantime	1279	2803	25268	386	29	29	0	37	37	0	34	1
44	siler	1124	1354	9518	175	0	0	0	0	0	0	0	0
45	laragon	2681	2532	49531	237	19	19	0	12	12	0	7	1
46	librenms	2801	14740	189156	597	39	103	64	969	990	21	158	6
47	Carbon-Forum	1822	3217	18462	113	33	38	5	85	85	0	20	3
48	Heimdall	4541	38854	500566	852	0	0	0	41	41	0	1	1
49	livehelperchat	1657	16291	259066	1026	455	801	346	615	791	176	335	3
50	Bonfire	1412	9957	94654	290	12	12	0	254	254	0	42	2
51	maccms10	1315	8759	79683	247	64	114	50	78	78	0	43	4
52	mailcow	5379	10489	117508	312	2201	2201	0	196	196	0	0	0
53	mantisbt	1430	4619	75258	101	48	113	65	273	286	13	197	4
54	matomo	16591	56514	278007	479	12	21	9	5	7	2	127	3
55	wp-heroku	1310	16100	364351	1579	1289	1447	158	214	271	57	671	4
56	microweber	2441	37534	223627	552	486	502	16	186	187	1	110	1
57	RPi-Jukebox	1027	689	7418	229	262	299	37	222	222	0	10	2
58	revolution	1303	13340	212276	261	4	5	1	72	96	24	36	3
59	Tieba-Cloud	1457	354	6752	106	155	161	6	154	212	58	41	2
60	nextcloud	19615	36516	374898	149	0	1	1	67	67	0	31	3
61	TeamPass	1423	36731	308672	190	100	102	2	1111	1112	1	46	2
62	php-saml	1041	3718	12467	126	0	0	0	3	3	0	29	1
63	opencart	6524	20323	140010	102	0	0	0	44	44	0	1	1
64	openemr	1950	43594	614196	5700	709	709	0	2165	2165	0	0	0
65	opnsense	2045	7011	90751	748	796	923	127	918	918	0	141	2
66	osTicket	2468	9694	186413	1372	152	569	417	546	549	3	284	2
67	owncloud	7785	141876	1142883	288	9	28	19	167	180	13	124	2
68	pfsense	3751	7660	144419	4501	841	944	103	312	330	18	382	3
69	codefever	2157	9893	84621	170	2	2	0	17	17	0	19	2

70	phabricator	12264	42138	507825	119	0	0	0	24	25	1	22	1
71	cphalcon	10593	12477	136374	454	0	0	0	0	0	0	0	0
72	phoronix	1717	3491	69063	444	320	360	40	734	739	5	64	3
73	phpbb	1545	10958	373623	260	5	6	1	1051	1054	3	55	3
74	phpipam	1722	8574	81371	2633	392	3447	3055	2611	2711	100	640	5
75	phpmyadmin	6103	16629	169239	3082	9	30	21	417	433	16	353	5
76	AdminLTE	1539	889	9289	222	7	7	0	40	42	2	9	1
77	Piwigo	1971	9214	165970	743	263	314	51	572	581	9	200	2
78	PrestaShop	6544	39985	388000	762	153	181	28	333	351	18	125	2
79	PrivateBin	4191	1637	11487	182	4	4	0	1	2	1	8	1
80	q2a	1533	3052	39491	110	66	73	7	43	57	14	125	3
81	raspap-webgui	3682	6620	8196	263	16	16	0	81	81	0	6	1
82	chevereto	2573	2095	24861	451	33	58	25	243	246	3	76	3
83	roundcubemail	4481	5990	76416	507	105	114	9	340	407	67	191	3
84	SuiteCRM	3053	40136	453028	7669	1496	1979	483	4157	4620	463	1021	6
85	skratos	2444	2765	27039	102	0	0	0	6	6	0	3	1
86	CMS-Hunter	1560	3770	48828	337	401	401	0	20	20	0	18	2
87	vesta	2672	2512	38648	1571	146	146	0	91	113	22	173	4
88	sw-platform	1925	41493	523192	294	0	0	0	149	149	0	0	0
89	shopware	1282	51066	355113	179	0	5	5	2094	2094	0	14	1
90	dokuwiki	3476	7869	223130	569	13	17	4	57	65	8	57	1
91	symfony	27127	40468	825093	400	0	0	0	78	78	0	3	1
92	testlink-code	1162	42148	461497	977	734	1023	289	409	502	93	780	4
93	ThinkUp	3316	9827	124552	768	182	349	167	75	94	19	290	2
94	WDScanner	1616	476	5443	193	172	172	0	125	138	13	11	2
95	thinkphp	2856	3768	51134	325	90	154	64	18	18	0	21	2
96	typecho-fans	1428	24394	91606	232	15	27	12	154	158	4	131	6
97	vanilla	2548	25780	220101	217	2	34	32	102	102	0	30	3
98	adminer	5328	1612	26199	357	116	147	31	239	302	63	167	3
99	wordless	1405	7686	70934	105	53	53	0	35	35	0	6	2
100	valet-plus	1527	579	6141	100	0	0	0	11	13	2	2	1
101	Gazelle	1729	2277	71583	1361	2362	2391	29	1224	1808	584	99	2
102	mediawiki	3046	53017	618354	210	5	6	1	41	63	22	74	1
103	woocommerce	8062	20420	252407	968	0	0	0	68	68	0	204	3
104	WordPress	16450	44715	287812	1806	192	222	30	149	167	18	273	4
105	custom-fields-pro	1112	5458	18635	123	0	0	0	1	1	0	0	0
106	PicUploader	1017	67475	688682	184	5	5	0	390	402	12	26	1
107	FruityWifi	2026	4099	18112	186	63	63	0	31	32	1	2	1
108	yii	4830	11056	743321	1015	2	7	5	21	21	0	13	1
109	yii2	13966	11888	117759	431	0	0	0	3	3	0	3	1
110	YOURLS	8268	2548	42477	179	44	61	17	39	50	11	62	3
111	pikachu	2264	204	6934	223	211	211	0	71	76	5	39	2
112	zoneminder	3744	14517	213544	1531	219	323	104	334	688	354	313	2
113	skycaiji	1578	17181	137508	251	44	102	58	148	156	8	97	4
114	dzzoffice	3499	28897	150297	328	501	2562	2061	679	780	101	322	5
	SUM	448K	1.9M	21.4M	85K	49231	61583	12352	50217	54985	4768	17308	251

Project info: Stars in Github, num of functions, PHP line of code, num of Sources. XSS, SQLI and FileM Discoveries: num of Comm_2 alerts (before, after and diff), num of Comm_1 alerts (before, after and diff). Our approach - other info: num of stitches and the iterations till no more stitches.