Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications

Feras Al Kassar*‡, Giulia Clerici*, Luca Compagna*, Fabian Yamaguchi†, Davide Balzarotti‡
{feras.al-kassar, giulia.clerici, luca.compagna}@sap.com, fabs@shiftleft.io, davide.balzarotti@eurecom.fr
*SAP Security Research †ShiftLeft Inc ‡EURECOM

***Abstract*—While static application security testing tools (SAST) have many known limitations, the impact of coding style on their ability to discover vulnerabilities remained largely unexplored. To fill this gap, in this study we experimented with a combination of commercial and open source security scanners, and compiled a list of over 270 different code patterns that, when present, impede the ability of state-of-the-art tools to analyze PHP and JavaScript code. By discovering the presence of these patterns during the software development lifecycle, our approach can provide important feedback to developers about the *testability* of their code. It can also help them to better assess the residual risk that the code could still contain vulnerabilities even when static analyzers report no findings. Finally, our approach can also point to alternative ways to transform the code to increase its testability for SAST.**

Our experiments show that testability tarpits are very common. For instance, an average PHP application contains over 21 of them and even the best state of art static analysis tools fail to analyze more than 20 consecutive instructions before encountering one of them. To assess the impact of pattern transformations over static analysis findings, we experimented with both manual and automated code transformations designed to replace a subset of patterns with equivalent, but more testable, code. These transformations allowed existing tools to better understand and analyze the applications, and lead to the detection of 440 new potential vulnerabilities in 48 projects. We responsibly disclosed all these issues: 31 projects already answered confirming 182 vulnerabilities. Out of these confirmed issues– that remained previously unknown due to the poor testability of the applications code– there are 38 impacting popular Github projects (>1k stars), such as PHP Dzzoffice (3.3k), JS Docsify (19k), and JS Apexcharts (11k). 25 CVEs have been already published and we have others in-process.

I. Introduction

According to the 2020 Edgescan Security Report, "*Web application security is where the majority of risk still resides*" [13]. This is confirmed by the fact that most of the recent data breaches took advantage of the poor security of web applications. From a defensive point of view, there are two main options to detect vulnerabilities in web applications: static application security testing (SAST) and dynamic application security testing (DAST). Dynamic approaches are sound, but often treat the application as a black box and are therefore severely limited in the number of vulnerabilities they can detect. Static tools can instead reason about the entire behavior of the application, thus potentially detecting more vulnerabilities. However, in practice, they are neither sound nor complete, and often result in very large amounts of false positives.

To mitigate this problem, a large amount of research has been conducted to improve these two numbers, by either proposing techniques to increase the ability of static analysis to discover more vulnerabilities or to reduce the number of false alarms. Despite the progress done in both directions, it is undeniable that SAST tools still struggle to cope with the complexity of real-world code – which is one of the reasons for the poor security of today's web applications.

In this paper, we look at the problem from a different angle. In particular, we focus on another limitation of these tools that is often neglected: the fact that it is very difficult (independently from the tool's precision) for an analyst to interpret their results. In the example above, how can we translate the lack of vulnerabilities reported by a SAST tool into an actionable insight on the security of the application?
A key observation that motivates our work is that while the precision of the results depends on the tool, the level of "*confidence*" largely depends on the application. For instance, if zero vulnerabilities are reported in a small application with only a handful of untrusted input, the analyst might be confident that the tool was right and the codebase could not contain many undetected vulnerabilities. In contrast, if the same result is returned on a very large and complex application, that confidence might be much lower, therefore resulting in a higher *residual risk* that the code could still contain vulnerabilities.

Our goal is to find a way to capture this residual risk by proposing a novel approach based on the concept of **testability tarpits**. A tarpit is a specific pattern of code that is known to cause problems for a class of static analysis tools. Other researchers have reported such patterns as a way to point out the current limitations

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24150
www.ndss-symposium.org

(and possible venues for improvement) of static analysis tools. We propose instead to use them as a metric to capture *how testable* an application is. The intuition is that the residual risk of undiscovered vulnerabilities is lower if the application was easy to analyze, and higher if it presented many challenges for the analysis tool.

By building upon a comprehensive library of testability tarpits, we propose a general framework that can support a more principled understanding of the results of one (or a composition of) SAST tool(s). Our approach does not only provide a way to assess the confidence of the reported results, but it also points to the precise nature and location of the code that reduces this confidence. As a result, an analyst can decide to add more tools to reduce the impact of the existing tarpits, to perform a manual audit of a poorly testable part of the code, or to refactor part of the application to increase its testability.

While the methodology we present is general and independent of the language of the application and the class of tools used to analyze it, in this paper we focus on static analysis tools for PHP and Javascript (JS, in short) code, the two most common languages for web application development. In particular, we create a *library* of 122 testability tarpits for PHP and 153 tarpits for JS (cf. III-A), covering language features, built-in APIs, security-related functionalities, and static and dynamic operations. We then selected an *arsenal* of 11 commercial and open-source SAST tools (6 for PHP and 5 for JS) and we assessed them against our tarpits' libraries (cf. Section III-B). The best commercial tools were only able to handle 50% of the PHP and 60% of the JS tarpits, thus potentially leaving large parts of an application code unexplored. To measure the impact on those unsupported tarpits, we implemented automated *discovery rules* (cf. Section IV) for all our PHP patterns and used them to scan 3341 open-source PHP applications. Our experiments (cf. Section V) demonstrate that these tarpits are very common in the real world: the average project contains 21 different tarpits and even the best SAST tool cannot process more than 20 consecutive instructions without encountering a pattern that prevents it from correctly analyzing the code.

The ability to automatically discover each tarpit brings many benefits. First, it can provide immediate and precise *feedback to the developers* about the tarpits in their code (e.g., by integrating the discovery rules into an IDE). This information can then be used to make an informed decision about which combination of SAST tools are better suited to analyze the code, which parts of the application are *blind spots* for a static analyzer and thus may require a more extensive code review process, and which region of code could be *refactored* into *more testable* alternatives.

We conclude our study by performing two experiments to assess the use of code refactoring as a mean to make an application more testable for SAST tools. In the first (Section VI), we *manually investigate* five PHP and five JS applications, for which SAST tools were unable to discover the presence of known vulnerabilities. By transforming the testability tarpits we enabled the tools to detect the vulnerabilities. Moreover, over 200 additional bugs were reported, leading us to the disclosure of 71 confirmed vulnerabilities, as some of the discovered issues still applied to the latest version of the tested projects. In the second experiment (Section VII), we target instead thousands of popular real-world applications (the same we used for the prevalence experiment), to which we apply five pattern transformations in a fully *automated* fashion. Our tool modified 1170 applications, by transforming 32,192 occurrences of the five tarpits. By running SAST tools both before and after the transformations we could observe the improvement in the overall testability, supported by the detection of hundreds of previously unknown vulnerabilities. In particular, we discovered 370 vulnerabilities in 43 different applications, 55 of which affected very popular projects with more than 1000 stars in Github. We responsibly disclosed all issues, and we have received 111 confirmation from the development teams (36 confirmations for the popular projects). These outcomes confirm the added-value of our approach and the impact of removing tarpits to increase testability for SAST tools.

All the testability patterns and the resources of this research (for both PHP and JS) are available in our (currently anonymized) repository [3].

```
1  // FILE: core/gpc_api.php
2  function gpc_get( $name, ..) {
3   if( isset( $_POST[$name] ) ) {
4    $r = gpc_strip_slashes( $_POST[$name] );
5   }
6   ...
7   return $r;
8  }
9
10 function gpc_get_string($name, ..) {
11  $args = func_get_args();
12  $r = call_user_func_array('gpc_get', $args);
13  ...
14  return $r;
15 }
16
17 // FILE: bug_actiongroup_ext.php
18 $act = gpc_get_string('action');
19 $act_file = 'bug_actiongroup_' . $act . '_inc.php';
20 require_once(.. . $act_file);
```

Listing 1. Example of a file injection vulnerability in MantisBT

II. Approach Overview

The source-code excerpt shown in Listing 1, simplified for presentation, highlights a file injection vulnerability (CVE-2011-3357) in the popular Mantis Bug Tracker. The code uses the function `require_once` to include and evaluate code from an external file (line 20). Care must be taken to ensure that users cannot freely choose the name of the file as this would permit them to execute arbitrary code. Unfortunately, in the example, the file name ultimately depends on the value of an unsanitized POST request-parameter, a value that an attacker fully controls. The example illustrates the complex interprocedural assignment chains that a static analyzer must correctly

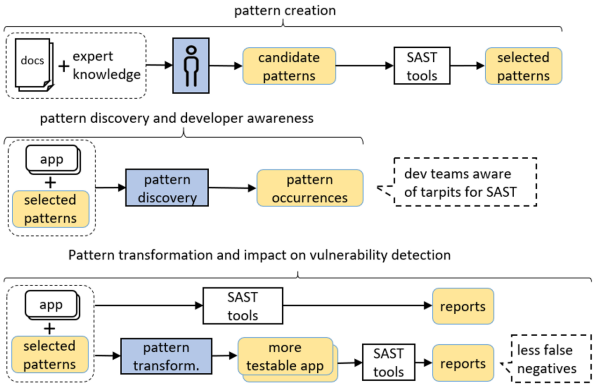

Fig. 1. Approach outline

handle in order to identify the vulnerability: the file name depends on the variable `$act` that is initialized via a call to the application-defined utility function `gpc_get_string` (line 18-19). This function internally makes use of the PHP functions `func_get_args` and `call_user_func_array` to dynamically invoke the variadic function `gpc_get` (line 12). Finally, `gpc_get` accesses the attacker-controlled POST parameter (line 3-5).

The majority of SAST tools in our selection (see Section III) are not able to detect this vulnerability. Through a manual investigation, we discovered that the dynamic function invocation (`call_user_func_array`) prevented them to connect the user-provided parameter to the name of the included file. In addition, some tools are also unable to handle the `func_get_args` function, which again affects the data-flow of the application.[1] Our finding is confirmed by the fact that a simple refactoring of line 12 into the equivalent `$r = gpc_get($args)` is sufficient to enable the tools to report the file injection vulnerability.

The main goal of our research is to build upon this observation and use the concepts of *testability tarpits* to build a new framework to assess the security of web applications. Our approach, outlined in Figure 1, is composed of three phases.

Pattern creation. The first objective of our work is to compile a comprehensive list of testability tarpits, that is, code patterns that impede the ability of SAST tools to reason about the code and identify vulnerabilities. The creation and selection process, which we describe in detail in Section III, involved an extensive manual effort. To show the applicability of our approach to different programming languages, we reviewed the documentation, the internal specifications, and the APIs of both PHP and JS and distilled this information into a number of code snippets that emphasize different functionalities. We then embedded these patterns in small test cases, which we tested on a set of commercial and open sources SAST tools (5 for JS and 6 for PHP) to identify the tarpits that could impede the testability of an application.

Pattern discovery and developer awareness. In the second phase of our approach, we implemented a tool to identify instances of our patterns in the source code of a program. As we strongly believe that extending existing production-quality tools trumps re-implementing basic code analysis from scratch, we base our tool on the code analysis platform Joern [40]. The platform allows code patterns to be formulated in a domain-specific query language that provides access to syntax, control flow, and data flow properties. While the platform offers built-in patterns for the discovery of vulnerabilities in C/C++ code, patterns for Web applications are only scarcely available and focus entirely on the discovery of vulnerabilities. Our work performed several improvements to Joern's core analysis engine and PHP support and we developed queries to allow the discovery of testability problems. Both queries and improvements were contributed back to the project, enabling Joern to discover not only vulnerabilities but also testability problems.

Our tool can discover the presence of testability tarpits and make developers aware of the exact snippets of code that will confuse the SAST tools in their arsenal. While until today developers were only aware that a given SAST tool did not discover any vulnerability in their code, our approach provides additional information that can be integrated into a risk assessment methodology or used, for instance, to select other SAST tools that are more suitable for a specific application. To evaluate our tool and measure the overall prevalence of our library of tarpits in real world applications, in Section IV we present the results of the experiments we conducted on 3341 PHP applications.

Pattern transformation and impact on vulnerability discovery. While awareness is very important, in the final phase of our process we discuss how the identified tarpits can be removed by transforming the corresponding code. Our goal is to show that by transforming the code, the testability improves and SAST tools become capable to uncover more vulnerabilities. We evaluate this idea by applying different types of transformation rules in two separate experiments. In the first (Section VI), we manually investigated ten applications (five PHP and five JS) for which SAST tools were unable to discover known vulnerabilities. In this case, we progressively refactored all testability tarpits until the tools were able to detect the bugs. By making the applications more testable, this also allowed the tools to discover new, previously-unknown vulnerabilities. In fact, on the refactored code of the ten projects, the SAST tools in our arsenal reported 503 new alerts, 224 of which corresponded to true-positive vulnerabilities. In the second experiment (Section VII), we targeted instead thousands of popular real-world PHP applications, to which we apply five pattern transformations in a fully automated fashion. By running SAST tools both before and after the refactoring we could observe the improvement in the overall testability, supported by the detection of hundreds

[1]Note that both functions are not library APIs, but core features of PHP.

of previously unknown vulnerabilities. In particular, we discovered 370 security issues in 43 different applications, with 55 of these vulnerabilities affecting popular projects with more than 1000 stars in Github.

III. Pattern creation and selection

The first step of our approach consists of identifying those code patterns that prevent SAST tools from properly analyzing an application code. It is important to stress that in this paper we are only interested in *code-related* patterns, and not in other forms of architectural or deployment aspects that can affect testability. For instance, plugin-based infrastructures are often difficult to analyze statically, and tools often struggle to deal with application state maintained across requests (which could, for instance, lead to stored vulnerabilities). While these are also very important aspects, and they constitute possible venues to extend our work, in the rest of the paper we restrict our focus to source code patterns only. Because of this, while the approach we present is completely generic, the actual patterns we identify vary from one programming language to another. To show the generality of our approach, we performed our study on the two most popular programming languages for web application development: JS (ES11) and PHP (v7.4.9).

On top of their specificity for different languages, testability tarpits may also differ from one SAST tool to another. In fact, what constitutes a problem for a testing tool may be handled correctly by a different product, and vice versa. Moreover, developers often use a combination of tools to test their applications, thus making the final selection of patterns specific to each development and testing environment.

Therefore, we first selected a representative set of SAST tools that includes both commercial and open source solutions. In particular, based on the results reported by other studies that compared existing tools [5, 8], for PHP we selected RIPS [12], PHPsafe [33], WAP [36], and Progpilot [37] as representative of open-source solutions, and two leading commercial products referred here as Comm_1 and Comm_2.[2] For JS we selected instead three commercial tools (Comm_1, Comm_2, and Comm_3)[3], and NodeJsScan [4] and LGTM [2] as open source alternatives.[4] While this choice reflects the current state-of-the-art techniques used to analyze PHP and JS applications, our approach can be easily applied as-is to any other set of static analysis tools.

In the rest of this section, we discuss how we built our library of testability patterns (Section III-A) and present the experiments we conducted to validate them on our SAST toolset (Section III-B).

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

[3]Comm_1 and Comm_2 are the same commercial tools used for PHP.

[4]Notice that LGTM is subject to a commercial license when used on projects that are not open source. This was not the case for our study.

A. Pattern Creation

Given a programming language, we want to identify code patterns that can affect the ability of SAST tools to detect vulnerabilities. To this end, it is important to understand how static analyzers detect web vulnerabilities in the first place.

While many techniques exist, a common requirement is the ability to identify how user-provided input is propagated and processed by the application. Static taint analysis provides this capability and is employed in particular to uncover injection vulnerabilities such as SQL injections (SQLi), cross-site scripting (XSS), and code injections (CODEi). While the actual mechanism used for vulnerability detection is orthogonal to our approach, this observation provides us with a simple way to identify patterns.

The idea is to use a small fragment of code, hereinafter the *stub*, designed to receive an input value from the user (in the form of a GET parameter) and simply write it back to the output.

```php
$a = $_GET["p1"];
echo $a;
```

```javascript
const parsed = route.parse(req.url);
const query  = querystring.parse(parsed.query);
var c = query.name;
r.writeHead(200,{"Content-Type":"text/html"});
r.write(c);
r.end();
```

These two snippets (respectively for PHP and JS) contain a very simple form of reflected XSS that is correctly identified as vulnerable by every SAST tool on the market. To create our patterns we routinely customized these stubs by adding new operations – which represent our candidate tarpits – as part of the data flow between the source (the GET parameter) and the sink (the `echo` and `write` invocations).

For pattern creation, we systematically inspected all the chapters of the PHP and JS language documentations. We also analyzed comments describing special examples and corner cases and reviewed all the internal language APIs. For PHP, we also went through the entire instruction set of its intermediate language and verified that there was not a single opcode that was not covered by one of our patterns. By following this procedure, we manually created hundreds of test cases, each one dedicated to showcase a different aspect of the language. However, most of these snippets have no impact on the analysis performed by SAST tools. Therefore, we filtered the list by retaining only the problematic examples. For this purpose, we used our selection of SAST tools as oracles. Each tool was used to scan all candidate snippets, and for each test we verified whether the tool was still able to detect the reflected XSS vulnerability. To be conservative, if *at least one* of the tools failed to report the vulnerability, we saved the corresponding test case in our testability tarpits library. Hereafter we detail the five main dimensions we used to categorize our patterns and to guide our pattern-generation process.

Core language features vs built-in internal APIs: We started our investigation by studying the language documentation and, for PHP, the (often undocumented) list of internal opcodes (i.e., the low-level instructions that are processed by the Zend engine). For example, while *references* are a common concept present in most programming languages, under the hood PHP operates on them by using seven different opcodes (e.g., `ASSIGN_REF` creates a reference to a scalar variable and `RETURN_BY_REF` returns a reference from a function).

For instance, we integrated the `RETURN_BY_REF` opcode [5] in our stub by producing the following snippet:

```
1 class foo {
2  public $v = 42;
3  public function &getV() {return $this->v;}
4 }
5 $a = $_GET["p1"]; $obj = new foo;
6 $myV = &$obj->getV(); $obj->v = $a;
7 echo $myV;
```

In line 7, variable `$myV$` gets bound to the variable `obj->v` returned from `getV`. As a consequence, setting `obj->v` to the source `$a` in line 8 makes also `$myV$` changing, leading to a XSS in line 9.

In total we identified 96 challenging patterns for PHP and 153 for JS. However not all of our patterns are related to features of the language, some are instead associated with the use of core library functions. In fact, internal API functions are typically written in C for better performance, and therefore it is difficult for SAST tools to check their code during the analysis. To mitigate this problem SAST tools often maintain a set of models that describe the relationship (in terms of taint propagation) among the input and output parameters of these functions [12]. For instance, the code snippet below captures the testability pattern created for *extract*, an internal PHP API generating variables dynamically from an array:

```
1 $aaa = $_GET["p1"];
2 $arr = array("A"=>$aaa,"B"=>"BBB");
3 extract($arr); echo $A . $B;
```

In this specific case, variables `$A` and `$B` are created and assigned to `$aaa` and to `"BBB"`, respectively. This pattern enables evaluating whether a SAST tool models the extract function and properly propagates the user-controlled input `$aaa` into the variable `$A`. In total, we identified 26 patterns in PHP and 22 patterns in JS that target challenging internal APIs.

Security related: We already mentioned that SAST tools often employ taint-based dataflow analysis to detect vulnerabilities: when a user-controlled input (referred to as *source*) flows into a sensitive operation (referred to as *sink*), without being processed by a *sanitizer*, that dataflow is reported as a vulnerability. We thus created some testability patterns to probe how good the SAST tool is in recognizing sources, sinks, and sanitizers. For instance, the following code snipped captures a pattern instance evaluating whether a SAST tool supports the sink PHP operation `exit`, which terminates the application execution and passes a message to the user:

```
1 $a = $_GET["p1"];
2 exit($a);
```

Similarly to the `echo` sink, `exit` can lead to XSS vulnerabilities. In total, we identified 16 patterns in PHP and 22 in JS in this category.

Static vs Dynamic features: Developers often rely on code constructs that cannot be fully analyzed statically, because their exact behavior can only be determined at runtime. In some cases, this might be required by the application and therefore it might be difficult to rewrite the code in a different way. But in other cases, these are used just for convenience, as the result of cut&paste operations, or to support functionalities that have already been removed from the code long before. For instance, the motivating example we discussed in the previous section uses the dynamic operation `call_user_func_array`, a form of *dynamic dispatching*. However, the MantisBT developers hardcoded a constant parameter, thus making the target function resolvable from a static analysis perspective.

To capture these differences, we define four dynamic categories for our code snippets targeting dynamic operations:

D1: the core parameter of the dynamic operation (e.g., the first parameter in `call_user_func_array`) is an hardcoded constant.

```
1 /* D1 */ call_user_func_array("Func", $b);
```

D2: the parameter is an expression whose value can be univocally computed statically via constant propagation.

```
1 $a = "FuncA";
2 /* D2 */ call_user_func_array($a, $b);
```

D3: the parameter is an expression whose value can only be partially computed statically. E.g., in the following example, only functions starting with `"Func"` can be called. This could be used by SAST to reduce the over-approximation needed to cover all the possible execution paths.

```
1 /* D3 */ call_user_func_array("Func" . $v, $b);
```

D4: the parameter is an expression whose value cannot be computed statically. While in the general case it is not possible to handle code belonging to this category statically, it is still important to measure the prevalence of these patterns to assess the testability of the code.

```
1 /* D4 */ call_user_func_array($f, $b);
```

Evaluating SAST tools against these four dynamic categories of increasing complexity allows us to measure

[5] PHP documentation - Return By Reference: https://www.php.net/manual/en/language.references.return.php

more precisely their behavior against these challenging dynamic operations. 52 of our PHP patterns and 52 of our JS patterns involves dynamic features.

Positive vs Negative Test Cases: To deal with dynamic features that cannot be computed statically, SAST tools need to choose between two possible approaches: over-approximate (e.g., by assuming all elements in an array are tainted when one of them is), or under-approximate (e.g., by ignoring all elements altogether). The first can increase the number of false positives, while the second solution can miss real vulnerabilities. To better distinguish among the two cases, we complemented the tests we developed for dynamic features with special tests that used the same dynamic functionality but without a vulnerability.

For instance, the following JS pattern (related to arithmetic operations on an array index) was reported as vulnerable by Comm_2 but not by Comm_1.

```
1 const parsed = route.parse(req.url);
2 const query  = querystring.parse(parsed.query);
3 var c = query.name;
4 array = ['a', 'b', c, 'd'];
5 index = 3; index = index -1 ;
6 r.writeHead(200,{"Content-Type":"text/html"});
7 r.write(array[index]); // print c variable
8 r.end();
```

This could be due to the fact that Comm_2 can compute the index value statically, or it could simply be the consequence of the fact that the two SAST tools might adopt different strategies to deal with arrays (over-approximating the first and under-approximating the other). To answer this question, we created a negative version of the same pattern, where line 7 is replaced by:

```
7 res.write(array[index-1]); // print 'b' char
```

Since Comm_2 reports a vulnerability also for the negative version of the pattern, we can conclude that it was indeed applying an over-approximation to deal with the array. In total, we retained 7 negative pattern instances for PHP and 20 for JS, for which the negative version was still reported as vulnerable by some of the SAST tools, indicating an excessive over-approximation.

Functional vs Object-Oriented: As we discuss in the related work (Section IX) several studies conducted by the software engineering community have discussed the poor testability of object-oriented code. In general, researchers have found that when developers use more object-oriented features, projects become harder to test (even though this in the literature normally refers to dynamic testing). Therefore we included 39 PHP and 40 JS patterns related to classes, methods, static methods, and properties. These patterns cover different OO aspects, such as object constructors, encapsulation, overriding, and inheritance.

B. Pattern Selection

Figure 2 and 3 summarize the results of our SAST tools against our libraries of 122 PHP and 153 JS tarpits. Due to

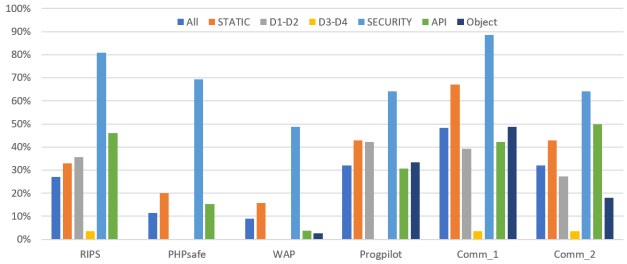

Fig. 2. SAST measurement over pattern dimensions (PHP)

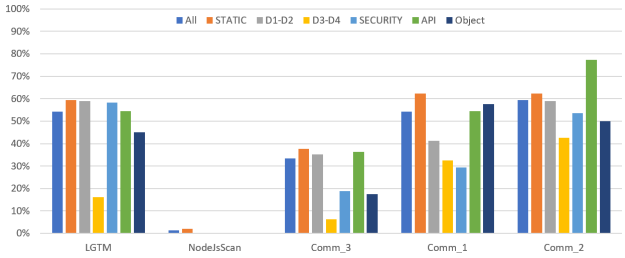

Fig. 3. SAST measurement over pattern dimensions (JS)

space limitation, the complete list of patterns is presented in Appendix and is detailed in our public repository [3].

In the graphs, the bars show the percentage of all patterns on which each tool reported the correct answer. First of all, it is interesting to observe that for PHP, none of the tools reaches 50% coverage of our patterns. Comm_1 is the best in terms of overall results, driven by its extensive coverage of PHP static features. However, other tools take the lead in other categories. For instance, Progpilot has the best support for the `D1-D2` tests (where more sophisticated static analysis algorithms might be required to propagate constant and determines the values of program variables), and Comm_2 has the highest coverage of the PHP APIs. RIPS is the best tool among the research tools for the supported APIs, confirmed by the fact that its authors mentioned in the corresponding paper [12] that they performed an extensive work to model the built-in function of PHP. On the other hand, Progpilot is the only research tool that supports object-oriented code, and it achieves the best results on static features between the open-source tools.

Results are a bit better for JS, where three tools are able to cover more than 50% of the patterns. However, in this case, the higher success rate is in part due to the extremely poor performance of NodeJsScan. Indeed, 24 patterns are supported by all tools but NodeJsScan, and therefore they would have been discarded if this tool was not part of our arsenal. As for PHP, commercial tools are still dominating, featuring similar performances and emerging as the best tool for the security dimension. If Comm_1 performs better on static features, Comm_2 outperforms it on the internal API support. With respect to dynamic patterns, we observe the same behavior as for PHP: tools supports some D1-D2 patterns, but have more troubles against D3-

D4 patterns. Only Comm_2 is able to correctly analyze beyond 40% of those D3-D4 instances, though it seems to apply over-approximation in most of those cases.

Another interesting aspect of our experiments is the fact that the tarpits are very different among the different tools. In fact, while tools taken individually have many limitations, their combination is able to handle around 66% of the PHP and even 85% of the JS patterns in our library. Moreover, none of the tools is a superset of any other in terms of tarpits. Thus, using a combination of SAST tools to test a web application is, from a testability point of view, always better than relying on a single product.

IV. Pattern discovery

In the previous section, we described how we built our library containing hundreds of testability tarpits. While this list can already be used to compare SAST tools and find a suitable combination that minimizes their limitations, this is not the main goal of our paper. Our objective is to support developers to assess which parts of their code can be effectively tested by SAST tools and which are not, and provide guidelines to improve the overall testability by avoiding particular patterns.

For this purpose, we decided to extend each tarpit in our library with a corresponding *discovery rule* that can be executed to discover its presence in the code of a Web application. While `grep`-like regular expressions can be sufficient to identify simple patterns, others require a more sophisticated taint-based analysis that takes into account the interplay of multiple program statements, e.g., to identify where a variable that is used in an interesting operation is previously defined. Designing a complete static analysis framework is beyond the scope of our paper, and therefore we decided to build our solution by extending the Joern code querying framework [40]. Joern constructs code property graphs (CPG, [43]) that combine the program's syntax tree, control flow, data dependencies, and calling relations in a joint representation – thus already providing most of the information needed for our analysis. While initially developed for the analysis of C/C++ code, the framework has recently been extended to handle PHP opcodes.[6] However, at the time of writing there was no public CPG generator for Node.js compatible with Joern, and this prevented us from performing a complete analysis of the prevalence of our JS patterns. Nevertheless, we scripted some ad-hoc routines to discover 54 of our JS patterns from the Abstract Syntax Tree (AST) of a JS application. Though these routines are not as precise as CPG queries and suffer from false positives, they have proved to be very helpful in pursuing initial experiments on real applications (e.g., to identify patterns before the manual transformation in Section VI).

Our Joern-based discovery rules have different complexities, ranging from a simple search for a given instruction to complex queries where we use the control, data, and call graph dependency. For instance, we use the following query to count the occurrences of the feature *simple reference* (`ASSIGN_REF`) in the CPG of a target application.

```
cpg.call(".*ASSIGN_REF.*").size
```

To find instead the use of objects in which the developer redefines the `__set` function, we can use Joern to search for the `NEW` opcode used on a class that has the method `__set` defined:

```
def hasSet = cpg.typeDecl
  .filter(_.method.name.contains("__set"))
  .name.l
cpg.call("NEW")
 .argument
 .filter{ x =>
    hasSet.contains(x.code.toLowerCase)}.size
```

The ability to automatically discover each tarpit brings many benefits. For instance, these rules can be integrated into an IDE to provide immediate and precise feedback to the developers about the impact of the code they are writing on the static testability of the application. This information can be used to make an informed decision about which parts of the application are *blind spots* for a static analyzer and thus may require a more extensive code review process, and which code patterns could be refactored into more analyzable alternatives. For instance, for security-critical services, developers may decide to limit the use of patterns that are not supported by SAST tools (e.g., those in the category D3-D4) to minimize the risk of undiscovered vulnerabilities.

V. Prevalence

We now estimate how prevalent our tarpits are in real-world applications. For this purpose, we use our CPG queries for PHP against four different datasets. The first three are composed of PHP projects hosted on GitHub, chosen according to their popularity (measured by the number of stars they received). In particular, we cloned 1000 applications of low popularity (between 20 and 70 stars), 1000 of medium popularity (between 200 and 700 stars), and 1000 with high popularity (more than 1000 stars).[7] We refer to the three datasets as G_L, G_M, G_H respectively. The detailed list of the cloned projects is available in our repository [3]. Finally, the fourth dataset consists of all applications from the Sourcecodester website (SC in brief), which hosts open-source PHP projects [42] that serve as references to other developers that want to implement their websites.

Due to space limitation, the complete results of our prevalence measurement are reported in Appendix (Table V). The whole experiment required 1 week on a 16-cores machine with 64 GB of memory. On average, for 1000 lines of code (LOC), generating the CPG took

[6]The extension is in an early development phase and yet to be released publicly. It was kindly made available to us by its developer—name removed for anonymity.

[7]We used Github-clone-all tool [1] to clone all these projects.

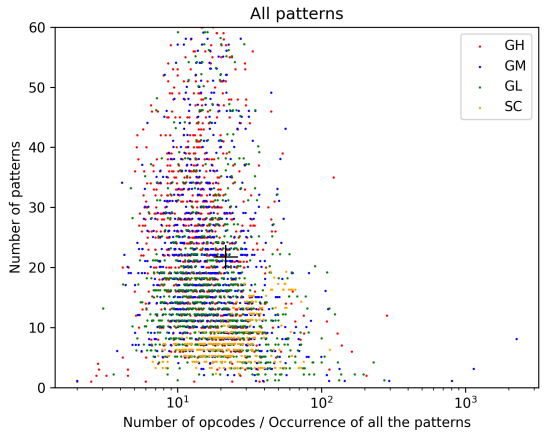

Fig. 4. Patterns distribution considering all patterns

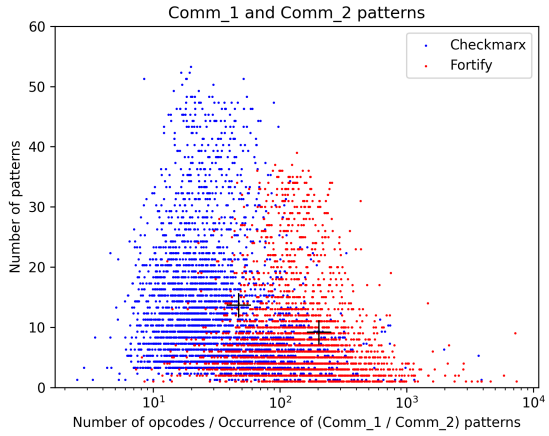

Fig. 5. Patterns distribution considering only patterns hard for Comm_1 and Comm_2

4.03 seconds, while traversing the CPG with our pattern discovery queries took 7.82 seconds.

The scatter plots in Figures 4 and 5 compactly present our results. In these plots, each dot represents a PHP application and its coordinates show the number of unique tarpits it contains (Y-axis) and the cumulative number of instances of such patterns normalized by the number of opcodes in the application (X-axis, plotted in log scale). We use the number of opcodes instead of the number of lines of code because it is more accurate as a line of code can contain multiple instructions and because opcodes represent only PHP code without considering other embedded elements such as HTML and JS. It is also important to note that Figure 4 does not show four tarpits: simple object (P21), simple array (P58), conditional assignment (P4) and combined operator (P5). In fact, these patterns occur with a very high frequency but only affects a few of our SAST tools.

Figure 4 shows the breakdown of the results by datasets. Here we can notice two important points. First, the prevalence of our patterns is very high: the average project contains 21.15 unique tarpits with an aggregated frequency of one tarpit every 8-to-50 opcode (the + sign in the figure represents the average point). Second, the more popular a project is, the more tarpits it tends to contain. The main reason is the size of the project that is higher for more popular projects: by counting the average number of opcodes in G_H, G_M, G_L, and, SC we have 62,303, 43,782, 21,066, and 17,141, respectively.

If we restrict our analysis to only those patterns that affect a given SAST tool, we observe marginal improvements. For instance, Figure 5 shows the results restricted to patterns that affect Comm_2 and Comm_1 for the three Github datasets. It is interesting to observe how the dots clearly follow different distributions, with blue dots closer to the upper-left corner and red dots closer to the bottom-right. This shows that real-world PHP applications are more difficult to test with the first tool than with the second: the average application contains 13.3 unique tarpits for Comm_2 (one every 47 opcodes) and 8.5 unique tarpits for Comm_1 (one every 203 opcodes).

Another interesting point is related to the scale of the Y-axis. For over 83% of the applications, the number of tarpits falls between one every 10 and one every 1000 opcodes – with an average of only 21. In other words, on real-world PHP applications even state-of-the-art SAST tools cannot process more than 20 consecutive opcodes without encountering a pattern that prevents them from correctly analyzing the code.

In the rest of this section we will discuss the prevalence of different types of patterns. To streamline the discussion, we only report numbers for the G_H dataset, but all values are available in the appendix for further analysis.

Object Oriented Code - 97% of the projects in the G_H dataset use objects. However, only one out of four of the open source SAST tools we tested supports object oriented code – showing a clear disconnection between research prototypes and real-world applications. By looking at OOP features that are not supported also by commercial tools, the magic methods `__set` (P32), `__get` (P33), and `__call` (P36) are used respectively by 6.1%, 8.1%, and 6.8% of the projects. This confirms that object oriented code still presents challenges also for the best SAST tools on the market.

PHP Features and APIs - If we exclude OOP, among all our tarpits the most prevalent language feature is the use of combined operators, which is present in 93.4% of the projects. However, this only causes problems to PHPSafe and is correctly handled by all other tools. If we look instead at those features that are causing problems for all our SAST tools, the most common are the use of static variables (71.2% of the projects) and raised exceptions (81%).

Regarding the built-in API functions, `array_map`,

which is not supported by any of our SAST tools, is present in 28% of the projects. Among the security critical sources and sinks, the use of superglobals (P66) and the `exit` function (P56) are the most common, with a prevalence of respectively 41.3% and 22.6% projects.

Dynamic Features - the dynamic features of a language are one of the biggest challenges for static analysis tools but also one of the most commonly used by developers. The simple case of storing and retrieving variables from dictionaries (P83), which often requires over-approximation from a static perspective, is used in 88.3% of the projects with a median of 32 occurrences each.

Features related to dynamic functions invocation are also among the most frequently used in our datasets. For instance, storing the function name in a variable (P82) is used in 448 projects, dynamic callbacks (P80) in 269 projects, and dynamic function call (P76) in 602 projects. All these examples belong to the D4 category – and therefore there is little a static tool can do to properly resolve these calls statically.

However, some of the dynamic patterns can be trivially handled by a SAST tool. For instance, the D1 instance of a dynamic call (P80), which hard-codes the name of the target to invoke, is present in 20.8% of the projects with a median of 2.5 instances each. Another simple example comes from file inclusion. In PHP, when file A includes file B, it can access the variables of B without defining them as global variables. The corresponding D1 pattern (P79) causes problems for three of our SAST tools (phpSAFE, WAP, and Comm_2) and it is present in 63.6% of the projects. The related and more complicated case where the file name is stored in a variable (D4), is even more common, with a prevalence of over 75%.

VI. Experiment: Manual Pattern Transformation

In this section, we will demonstrate the usage of our patterns on five PHP and five JS open-source projects selected according to these criteria: (i) the project is mainly based on PHP or JS, (ii) an injection vulnerability has been reported in the past for the project as a CVE precisely pointing to the latest vulnerable codebase version (*VulCode*, in short), (iii) some testability patterns occur in the path connecting the source and the sink of the vulnerability in *VulCode*, and (iv) one of the SAST commercial tools fails to report the CVE vulnerability on *VulCode*.

We gathered five projects for each language satisfying these criteria from the CVE Mitre website [30]. For each project, we carefully review the CVE and the testability patterns preventing the SAST tool from detecting the expected vulnerability. We then manually transformed these patterns and run again the tool to check if the vulnerability was detected on the transformed project code. Table I and Table II present the results of our experiments. For each project, we specify the CVE, the vulnerability class, the SAST tool used[8], the testability patterns identified and preventing the SAST tool from reporting the vulnerability, and the new injections (alerts) from the tool on the transformed code.

After refactoring the testability patterns, all the CVEs' vulnerabilities were detected and many more true positives were reported by the tools. Some of these true positives still applied to the latest versions of the projects and our responsible disclosure lead to three new CVEs[9].

In the rest of this section, we present some of the transformations that we applied and we discuss the new alerts reported by SAST tools. More details about the case study projects and their results are available in our repository [3].

A. Transformations

During these experiments, we encountered a combination of 9 unique PHP and 17 unique JS tarpits that prevented our tools (Comm_1 and/or Comm_2) from discovering the known vulnerabilities. To refactor the corresponding code patterns, we applied three types of transformations. We present hereafter one example from each type. The details for all transformations are available in our repository.

T1 - Semantic-preserving Transformations. This type of transformation can be applied automatically while preserving the semantic of the code. An example of this category is the refactoring of the D1 variant of `callback_functions` pattern (P78 in Table V) presented in Section II for the MantisBT PHP project. By replacing line 12 in Listing 1 with the equivalent `$r = gpc_get($args)`, the tarpit disappears, the code semantic is preserved, and the SAST tool becomes able to understand the code and detect the vulnerability. Overall, five out of nine (for PHP) and nine out of 17 (for JS) of our transformations belong to this category.

T2 - Over-approximations. Transformations in this category aim at reducing FNs by increasing the amount of code that can be analyzed by the SAST tool. However, they achieve this goal at the price of breaking the semantic of the program, as the removal of the tarpit requires to introduce an over-approximation. We experience this case in the Bakeshop Online Ordering PHP project where the pattern JS redirect (P57) is used to redirect the user to `index.php`:

```
1 function redirect($lc=Null){
2 echo"<script>window.location='{$lc}'</script>";
3 } redirect("index.php");
```

This pattern is not supported by any of the SAST PHP tools in our arsenal. To enable the SAST tool to understand the redirection we rewrote line 2 as `include($lc)`, thus removing the JS code altogether.

[8]For PHP projects, Comm_1 is the only tool that we used in this experiment and thus we avoid adding the Tool column.

[9]CVE-2021-33557, CVE-2021-33300, and CVE-2021-23342

TABLE I. IN-DEPTH ANALYSIS ON PHP REAL CASES

Project	CVE	Vuln.	Patterns	SAST After (TP/FP) XSS	SQLi	FILEi	CODEi	SUM
MantisBT	CVE-2011-3357	FILEi	callback_functions D1 (P80, T1)	28/88	0/0	18/14	3/0	49/102
Osclass	CVE-2012-0974	XSS	dynamic include D2 (P79, T1), array variable key D2 (P83, T2), static instance of a class (P49, T2)	14/0	0/0	0/0	0/0	14/0
Bakeshop Ordering	CVE-2020-35272	XSS	JS redirect (P57, T2)	15/0	0/0	0/0	0/0	15/0
Bus Booking	CVE-2020-25273	SQLi	Extract function (P70, T3)	0/0	49/0	0/0	0/0	49/0
Domainmod	CVE-2018-11404	XSS	Dirname function (P74, T1), Dynamic include D1 (P79, T1), buffer (P75, T1)	77/118	0/0	0/0	0/0	77/118
			9 patterns	134/206	49/0	18/14	3/0	204/220

TABLE II. IN-DEPTH ANALYSIS ON JS REAL CASES

Project	CVE	Vuln.	Tool	Patterns	SAST After (TP/FP) XSS	CODEi	SUM
Docsify	CVE-2020-7680	XSS	Comm_1	(P87, T1), (P49, T1), (P55, T1), (P79, T1), (P7, T1, T3), (P99, T2), (P78, T2), (P101, T2), (P21, T1)	5/24	0	5/24
Apexcharts	CVE-2021-23327	XSS	Comm_2	(P87, T1), (P101, T2)	6/3	0	6/3
Hello.JS	CVE-2020-7741	XSS	Comm_1	(P24, T1), (P82, T2), (P21, T1)	3/24	0	0
Lazysizes	CVE-2020-7642	XSS/ CODEi	Comm_2	(P14, T1), (P7, T1), (P83, T1), (P78, T2)	4/0	1/8	5/8
Angular Exp.	CVE-2021-21277	CODEi	Comm_2	(P87, T1), (P36, T2), (P86, T3), (P90, T2), (P75, T2), (P21, T1)	1/0	0	1/0
				17 patterns	19/51	1/8	20/59

However, this modification changes the semantics of the code as the redirection via `window.location` provides to the new page access *only* to the *session* variables of the previous page, while `include` provides access to all variables. For example, if there is a variable `v` controlled by the attacker in page A, `v` will be fully accessible to page B included in the context of page A. It is interesting to observe that this transformation introduces a new tarpit in the code: the `dynamic include D2` (P79) is occurring after the transformation as the location of the PHP file to be included is captured in a variable (`$lc`). However, this pattern instance is easily refactored via a T1 transformation. This shows that the refactoring cannot be performed in one single shot, but it needs to be repeated until no pattern instances are left in the code.

T3 - Developer-Assisted Transformations. While both T1 and T2 can be refactored in a fully-automated fashion, in some cases we noticed that we could not remove the tarpit without some form of human assistance. This support can be in the form of code annotations that specify the propagation of data within complex code areas. For instance, we encountered this situation in the Online Bus Booking PHP project. Here the developers use the extract function—which we explained in Section III-A—to generate variables dynamically from an array. SAST tools have trouble computing these variables statically. A simple annotation, added before the extract operation, is sufficient to help:

```
// @sast: propagate($_POST,[$username,$password])
```

Notice that this annotation can be added by the developers once our pattern discovery rules make them aware that SAST tools will not be able to interpret that pattern in their code and that some annotations would be helpful to overcome such SAST limitation. From that annotation is simple to execute a transformation that simply add to the code the following instructions to make the extract explicit:

```
$username = $_POST["username"];
$password= $_POST["password"];
```

Some SAST tools accept in input special annotations to help them in the analysis – and therefore this type of transformations could be implemented by using those annotations. However, we did not explore this direction in the paper as this approach would make the transformation tool-specific.

B. Results upon transformations

After running our SAST tools on the refactored code, we confirmed that they were able to correctly report all ten known vulnerabilities. Moreover, since the transformations allowed them to better analyze the application code, they also additionally reported 503 new potential injection vulnerabilities. After manual verification, we confirmed 224 of them to be true positives. In some cases, the new vulnerabilities were similar to the ones already reported in the CVEs. In other cases, after removing the tarpits the SAST tools were able to detect new vulnerabilities of different types. Motivated by the high number of new true positives, we moved our attention to the latest versions of the applications (*LatestCode*, in short). First, we used our discovery methodology (see Section IV) to confirm that the testability patterns preventing the detection of the old CVE were still present in the current code. Second, we run the testing tool on the latest versions, both before and after applying our pattern transformation to the whole project. This allowed us to discover several previously-unknown vulnerabilities, which received new CVEs. These include one in the MantisBT project (CVE-2021-33557), four in the latest version of Domainmod (CVE-2021-33300 in process), one in Docsify (CVE-2021-23342) and one in Apexcharts

TABLE III. LARGE-SCALE TRANSFORMATION EXPERIMENT

	SC		G_L		G_M		G_H	
	occ.	prj.	occ.	prj.	occ.	prj.	occ.	prj.
R_1	14	9	927	128	2029	160	2116	210
R_2	21	3	89	24	155	39	202	53
R_3	130	9	1287	125	1929	179	2582	254
R_4	130	12	3613	173	8724	263	7043	340
R_5	21	7	258	89	524	113	488	149
Total	316	21	6174	281	13361	382	12341	486
Alerts	18	3	7086	19	1019	26	1297	24
TP	18	3	224	13	73	12	55	15

(validated by the developers and rewarded with a bug bounty, while a CVE will be released soon). Finally, for both Bakeshop Online Ordering and Online Bus Booking, all true positives reported in the old versions were still applicable as we directly worked on their latest version for the known CVEs experiment. However, in this case, no new CVE was assigned because the new findings were considered as variations of the main CVE.

VII. EXPERIMENT: AUTOMATED PATTERN TRANSFORMATION

In the previous section, we investigated whether known vulnerabilities could be discovered by SAST tools once the obstacles related to our testability patterns are removed from the corresponding applications. After their code was manually transformed, we discovered that SAST tools were also able to discover a number of new, previously-unknown vulnerabilities – thus confirming the negative impact that our patterns have on static analysis.

Motivated by these results, we decided to develop a number of fully automated routines to transform the code of five simple PHP patterns.[10] We then applied these routines to all the applications in our four datasets (G_L, G_M, G_H, and SC, introduced in Section IV) and run both the original and the transformed code through Comm_1 (the top-performer tool in our assessment for PHP). Finally, we manually compared the alerts reported by the tool before and after the transformations to distinguish between false positives and real vulnerabilities.

The five testability tarpits we selected to transform for this large-scale experiments are:

[R1] Callback functions (D1), P80 – as we described in Section II, `call_user_func` can be used to invoke a function or an object's method by passing its name as parameter. For the simple case in which the target function is constant, we implemented three refactoring rules:

```
//Before
call_user_func("F",$x);
call_user_func($obj,"method1");
call_user_func_array("F",$args);
//After
F($x); $obj->method1(); F(...$args);
```

[10]For this experiment to be done for JS, we need to wait for a CPG generator to automatically and precisely discover patterns, cf. Section IV.

[R2] Callback functions (D3), P80 – This routine applies to a variation of the previous pattern in which the name of the function to invoke is obtained by concatenating a prefix string to a variable. In this case, our routine transforms the code by first retrieving all functions whose name starts with the prefix and then by invoking them inside `if` statements:

```
/*Before*/ call_user_func("Func_" . $x);
/* After */
if($x == "F1") {Func_F1();}
else if($x == "F2") {Func_F2();}
else if ...
```

[R3] Get arguments, P17: In PHP it is possible to define a function to receive an arbitrary number of arguments and then retrieve them from the code by invoking the `func_get_args` function. We transform this pattern to use the PHP variadic function instead (less problematic for SAST tools):

```
/*Before*/function F(){$y=func_get_args();}
/*After*/ function F(...$x){$y = x;}
```

[R4] Foreach with array, P58: PHP provides an internal function called `array_keys` which returns the keys of an associative array. Our routine transforms the use of this function in *foreach* loops as follow:

```
/*Before*/foreach(array_keys($arr) as $key){}
/*After*/ foreach($arr as $key => $value){}
```

[R5] Exit, P55: the `exit` function terminates the program and prints to the user the message it receives as a parameter. Therefore, if the message is controlled by the user and not properly sanitized, this can result into a XSS vulnerability. SAST tools do not take this potential sync into account. Our routine makes this explicit by performing the following change:

```
/*Before*/exit($value);
/*After*/ echo($value); exit();
```

Table III reports the results of our experiment. For each of the five tarpits, we counted the number of occurrences (which were transformed by our automated routines) and the number of projects that were affected. In total, our tool modified 1170 applications, by refactoring 32,192 occurrences of the five tarpits. As a result of these transformations, Comm_1 raised 9420 additional alerts for 72 of these applications (an average of 130.8 alerts per application). For 17 of these applications the new alerts related to more than one vulnerability class. Specifically, the new alerts applied to 103 pairs of application and vulnerability class, referred to as (app, vuln).

The verification of the new alerts required an intensive manual effort. If the pair (app, vuln) featured less than 50 new alerts (in total, 82 pairs), then all were inspected and classified as either true positive or false positive (587 alerts were classified in this phase). Otherwise, we sampled 20%

of the new alerts related to the pair (app, vuln), ensuring that alerts with different source-sync combinations were included. This resulted in the manual verification of 21 pairs and their 8833 new alerts. Finally, if any true positive was identified in this step, then the entire set of new alerts of the corresponding pair was inspected (this resulted in three pairs for which we inspected 425 new alerts). In total, we manually inspected 2700 new alerts.

Overall, this process allowed us to confirm 370 vulnerabilities in 43 applications, all of which were responsibly disclosed to the respective developers. However, not all maintainers answered our messages. For instance, out of the 55 vulnerabilities discovered in 15 popular GitHub projects, 36 (from 10 projects) were confirmed. In the G_M dataset, developers acknowledged while we received only two answers from the low popular projects G_L after we disclose 224 vulnerabilities in 13 applications. Based on these discoveries, three CVEs (CVE-2021-43673, CVE-2021-43682, and CVE-2021-43687) have already been assigned and many more have been reserved and will be published soon.

We also discovered 18 unique vulnerabilities (CVE-2021-44280) in demo code applications (SC dataset) used to showcase functionalities for other developers and, thus, suitable to be copied&pasted to speed up code implementation, with the subtle risk of porting the vulnerabilities in other applications.

False Positives Discussion

Table III shows that a large fraction of the new alerts, (approximately 2300 over 2700 alerts we manually inspected) are false positives. However, it is important to understand that this does not mean that our *transformations are the cause for those false positives*. Our transformations increase the amount of code that can be analyzed and tested by SAST tools. The more code is analyzed, the more likely the tool is to report findings, most of which are unfortunately false positives. Without our refactoring, the SAST tools are simply blind to those code areas.

Those blind code areas may be many and their impact on false-negatives significant. For instance, by inspecting the MantisBT project—part of the G_H dataset and presented in Listing 1—we identified other 9 functions that, as `gpc_get_string`, suffer from the callback functions (P80, D1) tarpit and invoke `gpc_get` in a (useless) dynamic fashion. These functions are called 769 times in 182 files.

A second important point is that all the five automated transformations applied in our experiment are semantic-preserving (T1) and as such they do not add any over-approximation. Thus, the large numbers of false-positives emerging in that experiment is essentially due to the ability of SAST tools to better understand the application code. To confirm this observation, we performed an additional experiment in which we manually inspected a few popular projects (in G_H) for which we received answers (and confirmed vulnerabilities) from developers. Our goal was to demonstrate that there is no direct relation between our transformations and the false-positives rate. Table IV shows the results. For each project, we indicate the ratio TP/FP before transformations, the number of transformations, and the ratio TP/FP of the new alerts raised on the refactored code. We can see that the ratio between the number of transformations and the new alerts is very diversified. Note that these results were validated with the development teams to improve the projects' quality. For instance, we submitted a pull request for Librenms (2.4k stars) that was promptly accepted to fix the detected vulnerability.

TABLE IV. FALSE POSITIVES EXPERIMENTS

Project	Before (TP/FP)	Transf.	After (TP/FP)
MantisBT (1.4k)	0/90	73	1/207
Cloudflare-CNAME-Setup (1.3k)	42/46	16	10/10
Librenms (2.4k)	49/135	144	1/2

While the above discussion is true for the transformations we used in our large-scale experiments, not all possible transformations have no impact on false-positives. For instance, T2 transformations could, as a side effect, increase the number of false-positives and as such they should be used with parsimony. We foresee the developers playing a key role in our approach deciding whether those transformations should be applied or not.

VIII. LIMITATIONS

Pattern Discovery. In Section IV we explained how our discovery rules first perform a static analysis of the application code. However, since the patterns we want to discover are by definition those that are problematic for static analyzers, this might seem a contradiction. In reality, the fact that our rules can discover the presence of a tarpit (e.g., the use of a given instruction in a certain context) does not necessarily imply that the underlying static analysis engine is able to correctly handle the pattern. For instance, our rules can detect that a piece of code performs a string operation even though the static analysis framework the rule is built upon cannot reconstruct the actual value of the string.

It is important to note that, while implementing sufficiently precise static taint analysis is simpler on an intermediate language, such as PHP opcode, some information (such as class inheritance) is lost in the compilation process or optimized out by the JIT compiler. In case our rules need access to this information (five patterns in total) we had to implement a simple ad-hoc text processing script to detect the tarpits at a syntactic level.

In other cases, our queries might under-count the number of instances of a pattern due to the difficulties of static taint analysis, in particular when tracking taint that is not passed from function to function via a method call. Similarly, patterns in the D3 category (where only part of a value is constant) could be implemented in many possible ways, but we only count when they rely on string concatenation and not, for instance, on sub-strings substitutions.

Because of these limitations, it is important to understand that the number of times a pattern is reported by our rules is a lower bound over the actual number of times it can be present in the code.

New Patterns. We tried to be comprehensive in our pattern catalogs by systematically inspecting all the chapters of the language documentation (for both PHP and JavaScript) and by including development community comments about language corner cases. However, we reckon that new features (e.g., from new widely used libraries) may emerge and result in new patterns. For this reason, we designed our approach to be extensible towards the addition of new patterns and we are developing an open-source framework where the community can add patterns by following a well-defined format. The procedure is described in detail in our repository [3].[11]

Once a developer has identified a new challenging code fragment, adding the corresponding pattern is a matter of a few hours of work. The most challenging part is coding the pattern discovery rule, which requires some knowledge in Scala and Joern. However, the developer can count on many examples in our catalog as well as on a broad and active community [40], which is another advantage of building our system on top of a popular framework.

For example, if a user suspects that the use of the PHP API function `substr` could be a potential tarpit for SAST tools, she will first adapt the pattern stub (cf. Section III-A) by adding a call to `substr` between the source and the sink:

```
$a = $_GET["p1"];
$b = substr($a,0,15);
echo $b; // XSS
```

Second, she would initiate the SAST tools validation scan (an operation fully supported by our framework) to collect the impact on the new pattern. Third, if one tool fails to discover the vulnerability on the stub, the pattern will be added to our collection and a discovery rule will need to be created. This can be easily done by tuning any of the discovery rule created for other API patterns (e.g., P59-P65 in Table V) via a simple replacement of the API function name with `substr`:[12]

```
cpg.call(".*INIT_FCALL.*
    ").argument.order(2).code("substr").size
```

Beyond Injection Vulnerabilities. While we focused on injection vulnerabilities (XSS, SQLi, Code Injection, File injection, Command Injection, and Path Manipulation), other types of vulnerabilities can be covered as a future work (e.g., Information leakage and/or improper error-handling attacks [41]). In fact, even though our tarpits focus on data flow-related challenges, many of them are quite general and also impact the analysis of the control flow of the application. A complete set of tarpits related to control flow can be an interesting follow-up for our work.

[11] https://github.com/enferas/TestabilityTarpits/tree/main/Docs/AddingPatternProcess.md

[12] `INIT_FCALL` opcode is used to call internal functions and the name of the function is the second argument.

IX. Related Work

Over the past two decades, researchers have developed many tools and techniques [see e.g., 16, 27, 28, 31, 35] to statically identify vulnerabilities in source code. Our research does not introduce a new vulnerability discovery techniques but rather focuses on the difficulties these tools face. The three research areas most closely related to our work are: software testability, studies of the limitations of static analysis, and comparisons of web SAST tools. In the following, we discuss related work in each of the three areas.

Software Testability. While *testability* of software artifacts has been the subject of a large body of research in the software engineering community, its definition remains unclear. In a recent survey, Garousi et al. [15] collected 33 different definitions of testability from different sources, finding the most common to be that given by ISO, which defines testability as the "*attributes of software that bear on the effort needed to validate the software product*". This captures a common interpretation accepted by software engineers, which sees testability as a measure of the number of test cases needed to test a program and/or of the difficulty of generating those cases.

In this paper, we instead use the term testability to measure the ability of static analysis tools to "understand" the code, with the goal of discovering security vulnerabilities. As such, our definition captures not the effort required to generate test cases, but the challenge of analyzing the code.

Despite the difference in scope, our work shares similarities with other studies in which the authors focused on either *improving* or *measuring* testability. In the first category, researchers have mainly studied source code transformations to improve automated test data generation [7, 9, 18, 19, 20, 22, 23].

On the measurement side, Garousi et al. [15] discuss 35 papers that present metrics to deal with testability. For instance, Gupta et al. [17] propose three fuzzy metrics for object-oriented software testability: Depth of Inheritance Tree, Coupling Between Objects, and Response For a Class. More recently (in 2020), Oluwatosin et al. [34] list 20 publications that measure testability of software design and categorize them based on whether they were related to Encapsulation, Coupling, Cohesion, Inheritance, Polymorphism, or Complexity.

While no previous study has systematically looked at patterns that prevent static tools from discovering vulnerabilities in web applications, previous work has already covered some of the aspects that can affect testability of programs written in dynamic languages. For example, Alshahwan et al. [6] list three categories that affect the testability of web applications (1) Forms, (2) Client-side

validation, and (3) Server-side manipulation. Bures [11] instead define two types of patterns that affect testability, the first one represents the anti-patterns, and the second represents the functional features of the front-end application. The same author also introduced a semi-automated framework to collect metrics for automated testability [10].

Challenges for Static Analysis. Our work also shares similarities with research on the limitations of static code analysis. For example, Landman et al. [26] analyzed the main challenges to perform static analysis on programs that use Java reflection. The authors' experiments show that for 78% of the projects in their dataset, reflection could not be resolved statically. They also provided suggestions for developers to analyze reflection code as well as improvements for static tool builders. Sui et al. [39] compared three tools that had difficulties in discovering vulnerabilities in JAVA applications because of the presence of dynamic features and reflection. Other papers discussed the challenges in analyzing dynamic proxy API [14].

In general, the use of dynamic features is the main challenge for static analyzers. Kyriakakis et al. [25] defined a number of patterns of PHP dynamic features and they counted their frequency in 10 projects, while Hills et al. [21] proposed a categorization of the PHP features and counted them in 19 projects. The authors also counted the prevalence of dynamic features and how many times they could be resolved statically, finding that 78% of the *dynamic includes* and 61% of the *variable variables* are statically computable.

Medeiros and Neves [29] recently published the first work that looked at the impact of *coding styles* on the accuracy of SAST tools (RIPS, WAP, and phpSAFE) applied to web applications. The authors define six scenarios of coding styles, with three vulnerabilities each. In all cases, they found that the tools identify true positives when the query source is defined closer to the sink and false negatives when it is defined farther from the sink.

Comparison of SAST Tools. Several studies have compared the accuracy and effectiveness of SAST tools. Nunes et al. [32] proposed a benchmark to compare five popular tools on their ability to discover SQL injections and XSS vulnerabilities in 149 WordPress plugins. Kupschs and Miller [24] studied the ability of two popular commercial applications, Fortify and Coverity, to discover 15 vulnerabilities in the Condor project. Each vulnerability was validated by hand, and classified by the authors according to its difficulty to discover (8 Difficult, 1 Hard, 5 Easy). Finally, Spoto et al. [38] uses object-sensitive taint analysis to build their static taint analysis for web applications in JAVA and compare their results with ten static analyzing tools.

Novelty. While other researchers have performed similar studies (but limited to only a handful of patterns) to identify challenges for crawlers and dynamic testing, to the best of our knowledge this paper is the first to evaluate web applications based on how easy they are to analyze for SAST tools. By using our approach developers can get a precise indication of the fraction of the application that a SAST tool will not be able to cover. This helps to put the results of static testing into context and to suggest which SAST tool is better suited to analyze the application. Finally, our study is the first to show the impact of code refactoring on vulnerability discovery for web applications.

Previous works (Kyriakakis et al. [25] and Medeiros and Neves [29]) provided inspiration for some of our tarpits, in particular for the dynamic patterns. On the other hand, our tarpits provide new metrics to compare between static security tools, and provide a number of novel findings which were never discussed in the state of the art. Finally, previous papers have compared SAST tools based on their accuracy of discovering vulnerabilities in real-world applications, while in our work we compare them based on the types of code patterns they are able to handle correctly.

X. Conclusion and Future Work

In this paper, we demonstrated that specific code patterns, which we call testability tarpits, are a major impediment for static analysis of real world web applications. In particular, we assembled a library of testability tarpits for the two most used web programming languages (PHP and JS) and we validated them by using a mix of state of the art open-source and commercial SAST tools. By defining discovery rules for these tarpits and applying them on thousands of open-source applications, we showed that these tarpits are widely used, indicating that nowadays static analysis has plenty of *blind-spots*. Finally, we performed two sets of experiments to to show that refactoring the code to remove tarpits has a significant impact on the alerts reported by SAST tools, leading to the discovery of many previously-unknown vulnerabilities.

We believe the framework discussed in the paper introduces a novel way to think about security testing. By shifting the focus from the testing tools to the code of the application, our solution allows to better assess the residual risk that a vulnerability is still present in the code after static testing. We contributed all our discovery rules to the popular Joern community, in the hope that other researchers and developers will extend our library of tarpits and adopt to assess the security testing of web applications. All other results and resources we developed about our research are available in our public repository [3].

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] Github-clone-all tool. https://github.com/rhysd/github-clone-all.

[2] LGTM. https://lgtm.com/, Accessed March 12, 2022. Version: v1.27.0.

[3] Repository of testability patterns and experiments. https://github.com/enferas/TestabilityTarpits, Accessed March 12, 2022.

[4] Ajin Abraham. Nodejsscan. https://ajinabraham.github.io/nodejsscan/, Accessed March 12, 2022. Version: v4.5.

[5] Areej Algaith, Ivano Alessandro Elia, Ilir Gashi, and Marco Vieira. Diversity with intrusion detection systems: An empirical study. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–5, 2017.

[6] Nadia Alshahwan, Mark Harman, Alessandro Marchetto, and Paolo Tonella. Improving web application testing using testability measures. In *2009 11th IEEE International Symposium on Web Systems Evolution*, pages 49–58. IEEE, 2009.

[7] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Inf. Sci.*, 178:3075–3095, 08 2008.

[8] 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.

[9] Achim Brucker, Lukas Bruegger, Paul Kearney, and Burkhart Wolff. Verified firewall policy transformations for test case generation. pages 345–354, 01 2010.

[10] Miroslav Bures. Framework for assessment of web application automated testability. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*, RACS, page 512–514, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Miroslav Bures. Metrics for automated testability of web applications. In *Proceedings of the 16th International Conference on Computer Systems and Technologies*, CompSysTech '15, page 83–89, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. 01 2014. Version: v0.55.

[13] Edgescan. 2020 vulnerability statistics report.

[14] 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.

[15] Vahid Garousi, Michael Felderer, and Feyza Nur Kilicaslan. A survey on software testability. *Inf. Softw. Technol.*, 108:35–64, 2019.

[16] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.

[17] Vandana Gupta, K. Aggarwal, and Yogesh Singh. A fuzzy approach for integrated measure of object-oriented software testability. *Journal of Computer Science*, 1, 02 2005.

[18] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[19] Mark Harman. Refactoring as testability transformation. pages 414 – 421, 04 2011.

[20] Robert Hierons, Mark Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48, 05 2005.

[21] 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.

[22] AbdulSalam Kalaji, Robert Mark Hierons, and Stephen Swift. A testability transformation approach for state-based programs. In *2009 1st International Symposium on Search Based Software Engineering*, pages 85–88, 2009.

[23] B. Korel, Mark Harman, Sunhwa Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. pages 10 pp. – 254, 12 2005.

[24] 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.

[25] 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.

[26] 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.

[27] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX security symposium*, volume 14, pages 18–18, 2005.

[28] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, Vancouver, BC, August 2017. USENIX Association.

[29] Ibéria Medeiros and Nuno Neves. Impact of coding styles on behaviours of static analysis tools for web applications. DSN 2020, pages 55–56, 06 2020.

[30] CVE Mitre. https://cve.mitre.org/.

[31] NIST. NIST - Source Code Security Analyzers. https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers.

[32] 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.

[33] 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. Version: DSN2015.

[34] Onilede-Jacobs Oluwatosin, Abdullateef Balogun, Shuib Basri, Abimbola Akintola, and Amos Bajeh. Object-oriented measures as testability indicators: An empirical study. *Journal of Engineering Science and Technology*, 15:1092–1108, 04 2020.

[35] OWASP. OWASP - Source Code Analysis Tools. https://owasp.org/www-community/Source_Code_Analysis_Tools.

[36] OWASP. OWASP WAP – Web Application Protection Project. https://securityonline.info/owasp-wap-web-application-protection-project/. Version: v2.1.

[37] progpilot. progpilot - A static analyzer for security purposes. https://github.com/designsecurity/progpilot. Version: v0.7.

[38] 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.

[39] 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.

[40] The Joern Team. Joern - The Bug Hunter's Workbench. https://joern.io, 2021. Retrieved July 2021.

[41] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In *PLDI '09*, 2009.

[42] Sourcecodester Website. Sourcecodester - free source codes. https://www.sourcecodester.com/php-project, 2021.

[43] 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.

APPENDIX

A. Testability patterns for PHP

The testability patterns for PHP are presented in Table V. This table groups together (by using horizontal lines) pattern instances which address similar aspects of the language and have the same response from SAST tools. For each instance (tarpit), the table reports its name, its properties with respect to the dimensions introduced in Section III-A, and the tools that are affected by it (by using a sequence of letters, R for RIPS, S for PHPsafe, W for WAP, P for Progpilot, X for Comm_1 and Y for Comm_2). When a tool handles the pattern by means of an over-approximation, we mark its name with an overline. For instance, the string $----X\overline{Y}$ means that a tarpit is handled correctly only by Comm_1 and via over-approximation by Comm_2. The last fours sets of columns report the prevalence of each pattern instance in our four datasets – as expressed by the number of affected projects (prj column) and by the median number of occurrences of the pattern (med column).

Finally, when the same pattern has multiple instances (e.g., to describe tests belonging to different dynamic categories) that lead to the same result, we group them and report their number in the number of instances (*#i*) column.

TABLE V: Patterns

ID	Pattern	#i	API	SEC	Dyn	OOP	Neg	Tools	SC		G_L		G_M		G_H	
									prj	med	prj	med	prj	med	prj	med
1	static_variables	1			S			------	50	13	443	4	635	7	712	14.0
2	global_variables	1			S			-S--XY	89	10	203	7	213	10	210	12.0
3	global_array	1			S			--WP-Y	33	6	138	4.0	162	5.0	179	7
4	conditional_assignment	1			S			R-WPXY	221	74	795	18	890	31.5	908	59.0
5	combined_operator	1			S			R-WPXY	335	170	919	33	942	64.0	934	97.5
6	coalesce	1			S			RS--XY	0	0	0	0	280	6.0	433	11
7	string_arithmetic_operations	1		✓	S			RSW-XY	277	10	523	6	636	9.0	707	11
8	simple_reference	1			S			-----X-	42	39	163	9	231	5	292	6.0
9	reference_argument	1			S			-----XY	208	14	387	7	399	10	486	9.0
10	return_by_reference	1			S			-------	19	11	83	4	95	4	132	4.0
11	foreach_with_reference	1			S			-------	41	7	182	3.0	238	4.0	321	4
12	make_ref	2			S		✓	---P-$\overline{Y}$	25	6	116	6.0	134	4.0	180	4.0
13	assign_static_prop_ref	1			S			---PX-	9	1.0	19	1	10	1.0	17	1
14	object_assigned_by_reference	1			S			-----X-	22	21	100	6.5	107	7	155	5
15	nested_function	1			S			-SWPX-	66	3.5	166	4.0	222	4.0	283	5
16	variadic_functions	1			S			-----XY	1	12	59	3	143	3	239	3
17	get_arguments	1			S			R-----Y	23	5.0	106	4.0	137	4	191	4
18	send_unpack	1			S			RS--X-	1	31	73	3	146	3.0	264	4.0
19	closures	2			D2			-----XY	36	1	543	7	733	11	782	25.5
20	use_with_closures	2			D2		✓	-----X$\overline{Y}$	25	1	321	4	524	5.0	614	12.0
21	simple_object	1			S	✓		---PXY	336	199	968	350.5	977	863	974	1536.5
22	assign_object	1			S	✓		--W-X-	30	3	138	4.0	212	4.0	325	4
23	object_argument	1			S	✓		-----X-	119	30	591	23	718	53.5	804	79.5
24	new_self	1			S	✓		-----X-	41	6	162	2.0	249	3	351	3
25	clone	1			S	✓		---PX-	41	6	147	3	238	5.0	338	5.0
26	late_static_binding	2			D2	✓	✓	-----$\overline{X}$-	13	1.0	165	4	279	5	386	8.0
27	get_called_class	1			D2	✓		-------	0	0	16	1.0	28	1.0	34	2.0
28	static_methods	1			S	✓		---PX-	119	17	792	29.0	865	61	898	126.5
29	static_properties	1			S	✓		---PX-	93	48	406	12.0	498	14.0	615	20
30	anonymous_classes	1			S	✓		-----XY	1	6	30	2.0	81	3	174	3.0
31	static_method_variable	2			D2,D4	✓		-------	1	1	23	1	51	2	68	2.0
32	set_overloading	1			S	✓		-----X-	0	0	44	6.0	50	7.0	61	8
33	get_overloading	1			S	✓		-------	1	1	56	6.5	70	7.0	81	8
34	isset_overloading	1			S	✓		-------	1	1	32	6.5	34	7.0	49	7
35	unset_overloading	1			S	✓		-----X-	0	0	24	5.5	23	8	29	7
36	call_overloading	1			S	✓		-------	6	1.0	47	4	59	7	68	7.5
37	callstatic_overloading	1			S	✓		-------	3	22	12	35.0	20	141.0	26	41.0
38	invoke	1			S	✓		-----X-	1	21	6	1.5	9	3	11	4
39	serialize_unserialize	1			S	✓		---P--	42	1.5	135	5	143	8	188	6.0
40	trait	1			S	✓		-----XC	87	6	800	13.0	881	26	907	43
41	self_methods	1			S	✓		---PX-	68	35	360	12.0	482	15.0	602	22.5
42	destructor	1			S	✓		-------	102	1	59	4	78	5.5	88	7.0
43	tostring_echo_object	1			S	✓		-----XY	9	12	76	14.5	86	25.0	108	16.0
44	verify_return_type	2			S	✓		---PX-	41	32	454	9.0	607	21	713	44
45	static_method_from_variable	1			D2	✓		---P-Y	23	3	98	2.0	166	4.0	235	4
46	object_to_array	1			D2	✓		-------	50	13	290	4.0	405	4	475	5
47	Overriding	1			S	✓		---PXC	62	11	569	16	698	24	750	49
48	construct_with_inheritance	1			S	✓		---PX-	64	4.5	471	7	590	11	680	17
49	static_instance	1			S	✓		-------	9	3	103	1	114	3	161	2
50	throw_exception	1			D2			-------	84	24	610	9.0	739	13	810	21.0
51	catch_exception	1			S			RS---C	95	6	466	4.5	612	6.0	687	11
52	try_catch_finally	2			D2		✓	R---$\overline{XY}$	3	6	39	2	86	2.0	197	2
53	track_error	1			S			---P--	135	20	333	5	407	6	521	7
54	generators	1			S			R---X-	5	1.0	57	2	113	3	204	5.0
55	goto	1			S			-----X-	3	8	68	8.0	95	4	155	5
56	exit	1		✓	S			RSWP--	261	3	182	2.0	185	4	226	4.0
57	JS_redirect	1			S			-------	151	8	27	3.0	115	3	137	4.0
58	simple_array	2			D1		✓	R--PX$\overline{Y}$	338	336.5	973	120	970	224.5	963	439
59	foreach_with_array	1	✓		S			R----$\overline{Y}$	68	9.5	208	4.0	237	5	297	4
	foreach_with_array	1			S			R--PX$\overline{Y}$	262	21	831	10	883	21	917	29
60	array_walk	2	✓		D2,D4			-------	19	1.0	43	1	60	2.0	74	2.0
61	array_map	2	✓		D2,D4			-------	41	12	167	3	214	5.0	280	4.0
62	parse_str_function	1	✓		D4			R-----	17	4.0	76	1.0	88	3.0	112	2.0
63	substring_replace_function	1	✓		S			RS---Y	38	4.0	80	3.0	97	3	118	3.0
64	preg_match	1	✓		S			R---X-	100	6.0	277	6	279	9	355	6

65	system (system)	1	✓	✓	S			R−−PXY	1	1	20	3.0	37	3	41	2
	system (exec)	1	✓	✓	S			R−−PXY	20	3	78	2.0	100	3.0	130	3.0
	system (unlink)	1	✓	✓	S			R−−PXY	101	3	159	4	181	6	237	5
66	superglobals	1		✓	S			−SWPX−	177	7.0	331	4	355	6	413	6
	superglobals	1		✓	S			R−W−X−	323	19	112	4.0	133	8	148	5.0
	superglobals	1		✓	S			−S−−X−	9	1	58	1.5	81	2	102	2.0
	superglobals	1		✓	S			RSWP−Y	240	9	90	4.0	120	4.0	124	4.0
67	odbc	1	✓	✓	S			RS−PXY	1	1	48	2.0	60	2.0	63	2
68	compact	2	✓		D2-D4			−−−−−−	1	1	48	2.0	60	2.0	63	2
69	create_function	1	✓		D1			−−−−−−	1	3	49	2	38	4.0	44	2.0
70	extract	1	✓		D2			−−−−−−	117	15	80	2.5	100	3.0	89	2
71	array_functions	1	✓		S			−−−−−$\overline{Y}$	23	1.0	86	2.0	114	2.0	155	3
	array_functions	1	✓		S			R−−−−$\overline{Y}$	10	1.0	23	2	37	1	40	2.0
72	procedural_queries	1	✓	✓	S			R−−PXY	187	26	30	2.5	33	4	24	4
	procedural_queries	2	✓	✓	S			−−−−XY	73	4	43	3	38	4	38	2.0
73	wrong_sanitizers	2	✓	✓	S			RS−PX−	174	5	242	3.0	303	5	332	5.0
74	dirname	1	✓	✓	D1			−−−−−−	23	3.0	101	4	111	4	131	4
75	buffer	1	✓		S			−−−−−−	27	2	150	2.5	190	3.0	181	4
76	function_variable	2			D2,D4			−−−−−−	40	2.5	315	3	465	4	602	7.0
77	object_callable	2			D2,D4			−−−−−−	12	6	89	2	141	3	252	4.0
78	autoloading_classes	1	✓		D2	✓		−−−P−−	50	1	123	1	138	1.0	150	2.0
79	dynamic_include	1			D1	✓		R−−PX−	337	44	549	5	600	5.0	636	6.0
	dynamic_include	1			D2	✓		R−−PX−	7	2	14	2.0	19	1	27	2
	dynamic_include	2			D3			−−−−−−	137	2	155	5	190	4.5	219	4
	dynamic_include	1			D4			−−−−−−	337	85.0	665	6	712	7.0	754	7.0
80	callback_functions	1			D1			−−−P−Y	41	2	128	3.0	159	3	208	2.5
	callback_functions	2			D2			−−−P−−	5	2	11	2	15	2	31	2
	callback_functions	1			D3			−−−−−−	8	7	17	2	29	1	38	1.5
	callback_functions	1			D4			−−−−−−	65	6	180	4.0	188	4.5	269	5
81	new_from_variable	1			D2	✓		−−−−−−	12	7	90	5.0	122	3.0	127	3
	new_from_variable	1			D3	✓		−−−−−−	0	0	0	0	0	0	0	0
	new_from_variable	1			D4	✓		−−−−−−	39	8	394	4.0	503	5	600	6.0
82	methods_variable	1			D2	✓		−−−−−−	14	4.0	99	4	163	4	211	4
	methods_variable	1			D4	✓		−−−−−−	46	3	223	3	351	3	448	4.0
83	array_variable_key	2			D2		✓	R−−−$\overline{XY}$	74	8.0	173	8	255	8	277	5
	array_variable_key	2			D4		✓	−−−−$\overline{XY}$	238	21	763	12	831	18	883	32
84	variable_variables	1			D2			−−−−−−	24	5.0	20	5.0	27	5	47	5
	variable_variables	1			D4			−−−−−−	123	5	81	4	108	4.5	147	3
	Total	122	26	16		39	7		7623	1716	22687	1031	27875	2004.5	32572	3097.5
	Average								74	16.66	220.2	10	270.6	19.5	316.23	30.07

Legenda for column Tools: RIPS (R), phpSAFE (S), WAP (W), Progpilot (P), Comm_1 (X), Comm_2(Y)

B. Testability patterns for JS

The JS testability patterns are listed and detailed for the community in our repository [3]. 34 of these patterns resemble their PHP siblings, targeting basic language constructs and operators (OOP, functions and variables). All the others focus on JS peculiarities such as specific data structure handlers (e.g., Proxy, WeakSet) or web operations (e.g., Ajax requests). We classified each pattern instance according to the same dimensions introduced in Section III-A for PHP: over 153 patterns instances, 40 are about *OOP*, 20 capture *negative test cases*, 22 are *security* related, and 22 refer to *internal API* (recall that these dimensions overlap between each other). For what concerns the *Static vs Dynamic features* dimension, 101 instances are static (*S*) and the rest are dynamic (17 belong to *D1*, 17 to *D2*, 10 to *D3* and 8 to *D4*). Table VI presents a selection of the patterns, with an emphasis on those mentioned in the paper.

TABLE VI: JS Patterns

ID	Pattern	#i	API	SEC	Dyn	OOP	Neg	Tools
7	array_unshift	1			S			L−−X−
14	template_literals	1		✓	S			L−ZXY
21	new_target	1			S			L−ZX−
24	finite	1			S			L−ZXY
36	returned_function	1			S			L−ZX−
49	arrow_function	1			D2			L−Z−Y
55	inheritance	1			S	✓		−−−−Y
75	functions_in_object	1			D1	✓		−−−−Y
78	asynchronous_event_handler	1	✓		D2			L−Z−Y
79	inline_function	1			D1			L−Z−Y
82	location_assign_with_search	1	✓	✓	S			L−Z−Y
83	getAttribute	1	✓	✓	S			−−−−Y
86	type_juggling	1			D3			−−−−−
87	modules	1			D2			−−−−Y
90	simple_array	1			S			L−ZXY
99	GET_ajax	1	✓	✓	D2			L−ZXY
101	innerHTML_outerHTML	2	✓	✓	S			L−−−Y
	Total	153	22	22		40	20	

Legenda for column Tools: LGTM (L), NodeJsScan (N), Comm_3 (Z), Comm_1 (X), Comm_2(Y)