From source code to crash test-cases through software testing automation

Robin David\textsuperscript{1}, Jonathan Salwan\textsuperscript{2} and Justin Bourroux\textsuperscript{3}

\textsuperscript{1}Quarkslab, 13 rue Saint Ambroise, Paris, France
\textsuperscript{2}Pirate, Atlantic Ocean, Earth
\textsuperscript{3}DGA-MI, Bruz, France

Abstract
Finding weaknesses and vulnerabilities in a source code is a difficult task. An approach to tackle this issue is static analysis. However, existing solutions and tools tend to generate numerous alerts and especially false positives. This paper presents an approach automating the software testing process from source code up to the dynamic testing of the compiled program. More specifically, from a static analysis report indicating alerts on source lines, it enables trying to cover these lines dynamically and opportunistically checking whether or not they can trigger a crash. The result is a test corpus allowing to cover alerts and to trigger them if they happen to be true positives. This paper discusses the methodology employed to track alerts down in the compiled binary, the testing engines selection process and the results obtained on a TCP/IP stack implementation for embedded and IoT systems.

Keywords
Software Testing, Static Analysis, Fuzzing, Dynamic Symbolic Execution, Vulnerability Research

1. Introduction

Context
Evaluating the security and finding flaws in source code is a tedious task in software testing. As a baseline, multiple guidelines have been published for a wide range of industries like automotive [1], aircraft [2] or aerospace [3] to identify weak and vulnerable code constructs. Applied for C code, the most known are MISRA C [4] and CERT C [5]. These standards are integrated in off-the-shelf static analyzers [6, 7, 8] which usually generate numerous alarms with substantially high false-positive rates. Therefore, analyzing results is a lengthy and cumbersome process. Few research in literature intend to solve the issue of validating alarms as generating a crashing or a violating test-case is an open research question. It requires solving both a reachability and a satisfiability issue in the program.

Our research does not address this issue directly but aims at bridging the gap between alerts identified at source level and the dynamic testing. That process aims at opportunistically covering and validating these alerts. We intend to automate the process as much as possible so that the analyst can focus on hard to reach corner-case alerts. This research is performed in the context of the PASTIS project (Programme d’Analyse Statique et de Tests Instrumentés pour la Sécurité) financed by DGA-MI which focuses on C, C++ programs and more specifically network related services.

Contributions
We present an automated testing infrastructure combining different testing techniques, namely fuzzing and Dynamic Symbolic Execution (DSE). Combining heterogenous testing engines to fuzz the same target is now usually called ensemble fuzzing [9].

We implemented our own fuzzing infrastructure and performed an experimental study of existing testing techniques namely fuzzing and DSE. We developed a benchmark test suite trying to reveal idiosyncratic behaviors of tested tools. Based on results obtained we selected 	extit{hongfuzz} [10] and 	extit{triton} [11], respectively for fuzzing and DSE. To summarize our research provides the following contributions:

- experimental study of existing techniques and tools on a dedicated benchmark;
- combination of a static analyzer with an ensemble fuzzer aggregating heterogenous software testing engines (greybox fuzzing and DSE);
- consolidation of this combination in a semi-automated workflow that starts from alerts on source code lines, track them back in the compiled binary and triggers automated...
testing to cover them and to trigger the bug if any. That process leads to the generation of a test corpus;
• benchmark assessing the robustness of two TCP/IP stacks which enabled uncovering a remote Denial-of-Service (DOS) which got assigned the identifier CVE-2021-26788.

2. Experimental study of techniques and tools

2.1. Software testing techniques

In the past decade, fuzzing [12] and dynamic symbolic execution [13], two software testing techniques, have revealed themselves as being very efficient at detecting and triggering bugs. While fuzzing tends to be very fast it can be hindered by some code constructs preventing it to progress in program exploration. Contrarily, DSE reasons more precisely on a per path manner but is significantly slower. Hence, we assessed various fuzzers and DSE engines to select one candidate of each to be combined together. Criterias and methodology are described in Section 2.2.

Fuzzing is the mean of feeding pseudo-random inputs to the program in order to trigger unexpected behaviors. Inputs can be generated randomly or using some feedback mechanisms. The most commonly used feedback is coverage but other feedbacks have been proposed in literature [14]. So-called greybox fuzzers like AFL [15], libfuzzer [16] or Honggfuzz [10] uses compilation-time static instrumentation of the program to obtain feedback at runtime.

AFL and Honggfuzz are two leading implementations of greybox fuzzers (now superseded by AFL++ [17]). AFL/QBDI enables binary-only fuzzing by interfacing AFL with QBDI [18]. This combination also enables on the fly optimizations, for instances, breaking comparisons with constants which are notoriously hard in mutational approaches. PULSAR has been selected for its availability to test network protocols. Input generation based on models (partially inferred) in comparison to AFL and Honggfuzz which are using genetic algorithms [19].

Dynamic Symbolic Execution also called whitebox-fuzzing uses a modeling of instruction semantic behavior to perform its execution. Instructions are disassembled and lifted in a semantic representation called intermediate representation used for emulation. A path $\pi$ in the program is then represented as a first-order logic formula (usually on bitvectors) that is then given to an SMT solver [20]. A solution of this formula is an input covering the path $\pi$.

<table>
<thead>
<tr>
<th>Version</th>
<th>manticore</th>
<th>KLEE</th>
<th>angr</th>
<th>Triton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>C</td>
<td>Py</td>
<td>C++</td>
<td>C</td>
</tr>
<tr>
<td>Open-source</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Base</td>
<td>binary</td>
<td>source</td>
<td>binary</td>
<td>binary</td>
</tr>
<tr>
<td>Intermediate Repr</td>
<td>custom</td>
<td>LLVM</td>
<td>VEX</td>
<td>custom</td>
</tr>
<tr>
<td>Library calls</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Syscalls</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Symbolic mem. read</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Symbolic mem. write</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bit-vectors</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Arrays</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2
Comparison of selected DSE tools

Table 2 shows DSE engines tested in this study. Both manticore [21] and angr [22] are developped in Python and provide similar features. They implement a wide range of library calls and syscalls, and support to some extend symbolic reads and writes in memory. Triton [11] provides more elementary functionalities but is designed to be modular in order to be embedded in a whole set of other utilities. KLEE [23] works on LLVM and is the reference in DSE.

2.2. Methodology & Benchmarking

To bring out two final candidates for fuzzing and DSE, we designed a test suite. It enables checking specific behaviors on small snippets (atomic tests) as well as testing the scale on larger programs. Atomic
tests assess the behavior, on symbolic pointers, handling of non-deterministic instructions and a variety of vulnerability categories (buffer-overflow, integer-overflow, use-after-free etc.). For scalability benchmark, uniq and base64 binaries of the LAVA-M project [24] have been used. This suite provides the ground truth along with some quantitative results (72 bugs in total in the two binaries).

To smooth statistical discrepancies of results caused by the random nature of fuzzing, tests were run multiple times and the mean value was computed. Each 70 atomic tests were run 3 times for a maximum duration of 300 seconds while scale binaries were run 3 times for 6h. Table 3 shows synthetic results of all utilities on this test suite. Every tools have been configured opportunistically with the best parameters to provide them fair chances. The PULSAR results have been excluded because it was not possible to run it correctly on the benchmark targets due to its network protocols focus.

<table>
<thead>
<tr>
<th></th>
<th>Atomic (70)</th>
<th>Scale (72)</th>
<th>Total (142)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>48</td>
<td>0</td>
<td>48 / 142</td>
</tr>
<tr>
<td>Honggfuzz</td>
<td>54</td>
<td>44</td>
<td>100 / 142</td>
</tr>
<tr>
<td>AFL/QBDI</td>
<td>47</td>
<td>33</td>
<td>80 / 142</td>
</tr>
<tr>
<td>manticore</td>
<td>34</td>
<td>0</td>
<td>34 / 142</td>
</tr>
<tr>
<td>KLEE</td>
<td>47</td>
<td>1</td>
<td>48 / 142</td>
</tr>
<tr>
<td>angr</td>
<td>37</td>
<td>0</td>
<td>37 / 142</td>
</tr>
<tr>
<td>Triton</td>
<td>47</td>
<td>0</td>
<td>47 / 142</td>
</tr>
</tbody>
</table>

Table 3
Test suite benchmark results

Fuzzing results shows that honggfuzz outperformed other engines and it has consequently been kept as the reference engine for fuzzing. For symbolic execution, while klee [23] outperformed other engines Triton has been selected. Being developers of Triton, the code is familiar to us, which makes it very easy to extend it and to modify it for PASTIS needs. Also, KLEE comes with two main issues for combining it with other fuzzing engines. First, as it works at LLVM-IR level it requires the program at source-level3. Then its code-base is evolving fast and contains many research-related features making it difficult to integrate it in a fully-automated workflow.

3. Testing Automation

3.1. Overview

The process of automating the dynamic testing of a source code is depicted in Figure 1. First, the code has to be harnessed4 to target the components of interest. It has to be prepared for both fuzzing and symbolic execution, which both have to be compiled differently. That step is highly manual and usually requires a good understanding of the target. Then the harnessed code can be provided to a Source Code Analysis Tool (SAST) that will generate a report of suspicious lines of code. These data are used to embed intrinsic function5 calls in the target in an automated manner. The code is compiled and provided to both testing engines that will intend to cover faulty lines and to generate crashing inputs. During that process they will communicate together to help each other. The final output is a report of alerts, indicating whether they have been covered or not and whether a crash has been associated to it. Automating most of these steps enable pruning some alerts enabling the analyst to focus on deepest uncovered ones.

For the purpose of this research, a fuzzing campaign is expected to run for at most 24h. That time cap has arbitrarily been set at the beginning of the PASTIS project.

3.2. Collaborative architecture

A challenge in designing an automated workflow is making the fuzzing and DSE to collaborate together and determining what kind of information to exchange. As both of these approaches work rather differently and have different notions of coverage, exchanging this kind of information directly is inherently complicated. Moreover, it makes difficult integrating new engines. Hence, each engine solely exchange input seeds they generate with regards to theirs own coverage metric. The remote engine is in charge of deciding whether it is valuable to keep an input or not. The exchange medium is described in Section 4.

The communication is performed through a central authority called broker which enables connecting multiple instances of the engines. Figure 2 shows the general overview of the collaborative architecture. At startup, the broker provides the binary with appropriate parameters to the engines. Then during the execution it forwards all the inputs re-

---

1Versions of tools are slightly outdated has the experiment was performed in 2018.
2While it is the case in this study, our goal was to make the PASTIS framework applicable to binary-only targets.
3Explaining the process is left out-of-scope for this paper.
4Ad-hoc function added in a code base, that will receive specific processing at runtime by a third-party tool.
From Source Code to Crash Test-Case through Software Testing Automation

Figure 1: Full Analysis Workflow

Initial Configuration
- binary
- SAST report (klocwork)
- configurations (coverage strategy, etc)

Master pastis-broker
communication
(Figure 1)

Workspace
- corpus / crashes / hangs
- log and client statistics
- CSV of results

Initial Configuration
communication
(Figure 1)

Master pastis-broker
- pastis-broker
- pastis-triton
- Pastis-DSE

Workspace
- corpus / crashes / hangs
- log and client statistics
- CSV of results

Figure 2: Global Collaborative Architecture Overview

received from one engine to the others. During the fuzzing campaign, if an engine covers or validates an alert it sends its identifier and the associated input to the broker that centralizes all data.

Alert Validation To validate alerts discovered at source-level they have to be trackable down in the compiled binary for the test engines. Compiling binaries in debug mode enables tracking the associated line of code for each assembly instructions. However, that requires each engines to be able to leverage debug information.

Listing 1: Intrinsic function

```
#define QB_INTRINSIC
int __klocwork_alert_placeholder(int id,
    const char* fmst, ...){
    printf("REACHED ID %d\n", id);
    return id;
}
#endif
```

Hence our workflow uses intrinsic functions. The Listing 1 shows the intrinsic function used. It takes an identifier, a format string and an arbitrary number of values as argument. Its sole purpose is printing the identifier given in parameter. Then, at each alert location a call to this function will be added with an unique identifier, the type of issue identified by the static analyzer and contextual parameters. For instance, it enables retrieving sizes of buffers that is known to the compiler but lost once compiled.

At runtime, a test engine will either have to parse `stdout` to find covered alerts, or to directly hook the intrinsic functions (depending on its inner-working). When detecting a crash or violation, engines are in charge to map it to a previously covered alert if applicable. Tracking the root-cause of a crash is still an open research problem [25] thus it is done here in an empirical manner. The last encountered alert is considered to be the cause of the crash and is thus considered validated. Note that we cannot invalidate an alert as being a false positive because of the potential infinite numbers of paths leading to that code location (path combinatorial problem).
Crashes or violations are detected in a different manner between fuzzing and DSE. Modern fuzzers use sanitizers like ASan [26], UBSan [27] or else TSan [28] that respectively detect: memory corruptions, undefined behavior or race-conditions. DSE engines usually implement their own sanitizers that leverage the analysis precision of symbolic execution to implement fine-grain sanitizers. In this case, the sanitizers can also use contextual information provided as argument of intrinsic functions to implement their checks.

4. Implementation

Target Setup Once the harness of the target program is implemented for all fuzzers included in the platform, the code is given to the SAST tool. In this research, the software Klocwork [7] developed by Perforce has been used. As output of its analysis, it provides an HTML file indicating faulty lines along with some additional contextual data (variable names, buffer sizes ...). Figure 3 provides an example of such report.

Our semi-automated workflow takes the report in input, translates it in JSON for easier processing and uses the result to automatically add intrinsic function calls in the source code. This code addition is made syntactically on a per line basis and thus requires to be double-checked by the analyst. Then the various variants of the program are compiled for each engines or target architecture (x86_64, ARM). The target program is now ready to be tested using the PASTIS framework.

PASTIS Framework The main interface with the analyst is the broker called pastis-broker. It is implemented in Python and ensures all communications between engines. The communication protocol is based on the message-queueing framework ZMQ\(^7\) so that it is interoperable with almost all existing programming languages.

The analyst launches pastis-broker with all the target binary variants, an initial corpus if needed, some configuration parameters, and the klocwork report to stop the campaign when all alerts are covered or validated.

Then the various test engines have to be launched with the broker IP address to receive all the fuzzing campaign data. If an engine supports different coverage strategies (block, edge, path etc) and multiple instances are connected to the broker, it will automatically equilibrate the coverage strategies.

Honggfuzz Integration Honggfuzz [10] is a modern greybox fuzzer developed in C++. Besides, being very efficient on many targets it has not been designed for collaborative fuzzing. As a consequence, small modifications have been made on its core. The most important is the ability to receive new inputs while it is already fuzzing\(^8\). Thereupon, a Python wrapper has been developed to perform all communications with the broker, to parse stdout, to inject external inputs received and to send the broker all inputs generated.

Such overlay is called a driver, as it enables interfacing an existing engine to the PASTIS framework. Figure 4 summarizes the main interactions between Honggfuzz and the wrapper with which all inter-communications are performed through filesystem monitoring (inotify on Linux). The whole component is called pastis-honggfuzz.

Triton Integration Triton [11] is a DSE framework library designed to perform symbolic execution on a given path. The whole logic of loading the program, scheduling input seeds, covering different inputs...
paths is left to the developer. To address this issue, a fully-featured DSE engine called TritonDSE has been developed at the top of Triton. For the purpose of the PASTIS framework, a program called PastisDSE has also been built on top of TritonDSE. This program performs all the communications with the broker which include receiving external inputs and sending ones generated by Triton. Figure 5 summarizes interactions of these components within the so-called pastis-triton component.

That component implements code, edge and path coverage strategies. It also implements different sanitizers for each category of vulnerability considered. As such, memory operations are tracked at the bit-level, and it enables detecting precisely off-by-one (OB1). Use-After-Free are detected by tracking the malloc and free primitives.

In this setting, pastis-triton is launched in pure-emulation (thus not as a concolic engine) to better control all side-effects and to allow the execution of Aarch64 binaries on x86 hosts. In essence, it has to emulate all the side-effects performed on the system (libc functions, syscalls etc). As it cannot be exhaustive, it only supports a limited number of libc functions and syscalls.

5. Experimental Results

5.1. CycloneTCP target

While this technique is applicable to any kind of software, the PASTIS project is centered on testing low-level network TCP/IP stacks. Among existing open-source implementations, CycloneTCP\(^9\) developed by Oryx-embedded, provides an implementation for a wide variety of protocols. Recent publications have shown it to be robust in comparison to other TCP/IP stacks [29].

The stack provides a driver mechanism to receive network frames for various MCUs and OSes. The target program is a simple HTTP server with a single static page. Only standard protocols are activated (Ethernet, IP, TCP, HTTP, ARP) in the target. Other protocols like, DNS, LLMNR, MBNS are not activated to focus on assessing the ability of engines to handle full TCP communications. The harness implements a driver which reads input frames from a file. A single input is thus a sequence of frames representing incoming messages from a client. The harness also tears down the multi-threading logic into a single threaded application enabling processing network frames in a sequential manner. While it prevents finding potential race-conditions it strongly reduces non-reproducible test-cases. As part of the harness, various patches were made in the code to remove checksums verification, add a pre-registered ARP lease (for the client) and to remove randomness of TCP Initial Sequence Number (ISN).

5.2. Controlled environment

To assess the workflow effectiveness, defects and vulnerabilities have been added to the CycloneTCP code. Defects are code constructs raising a SAST alert but which are structurally not triggerable, and vulnerabilities are defects that can be triggered. Such controlled benchmark enables checking the effectiveness of the framework to cover secluded locations of the code and to trigger vulnerabilities by creating the appropriate test-case (input).

Adding relevant defects is tedious as they have to fulfill the following properties:

- reachability: they have to be reachable by a test-case
- conditionality: they should be reachable under some conditions (not covered systematically)

---

9https://www.oryx-embedded.com/products/CycloneTCP
Within 24h, all intrinsic function calls correspond-
whole code lines. While it seems low, it repre-
Table 4 shows coverage and detection results.
work then have to cover and validate alerts within 
unsigned integers.
generates multiple false positives as the engine does 
ate faulty format strings (e.g %s). Conversely, 
harder to trigger as it requires the engine to gener-
ficulty varies. For example, FMT appeared to be 
control that automatic insertion of intrinsics does 
not break the program semantic.

To diversify vulnerabilities, 5 types are considered: 
BoF for buffer-overflow, IoF for integer-overflow, OB1 
for off-by-one, FMT for format-string (handling user-
put as format), UaF for Use-After-Free and SIGS 
for memory corruption (null pointers dereference 


to craft DHCP header, managing to enter 
exhaustiveness of the coverage (e.g: managing 
to enter HTTP parser, IPv4 reassembly etc)

5.3. Results

Table 4 shows coverage and detection results. 
Within 24h, all intrinsic function calls corresponding 
to identified alerts have been covered and 77% 
of vulnerabilities correctly validated. Multiple vul-
nerabilities are validated in less than a minute and 
and few of them took more than 3 hours to be detected.

The generated test corpora covers 42% of the 
whole core lines. While it seems low, it repre-
Sent almost all the coverable code. The rest being 
client-side functions only called when being used 
as a client. Besides that, the code is written in a 
defensive manner which implies that multiple error-
handling code are never covered. For instance, code 
malloc errors is never called as no out of 
memory were triggered. Quantitative results and 
experiments revealing the improvement of combing 
both testing engines have not yet been evaluated 
and is left as a future work.

Depending on the class of defects, validation diffi-
culty varies. For example, FMT appeared to be 
harder to trigger as it requires the engine to gener-
ate faulty format strings (e.g %s). Conversely, IoF do 
generates multiple false positives as the engine does 
not know if the operation is performed on signed or 
unsigned integers.

As part of the testing, many test-cases were caus-
ing the program to hang forever. While it strongly 
reduced the fuzzing speed it revealed to be a true 
0-day in the parsing of TCP options. It has responsi-
ibly been disclosed to Oryx-embedded which quickly 
published a patch. The vulnerability obtained the 
CVE identifier CVE-2021-26788\cite{CVE-2021-26788}.

5.4. Limitations

Most of the analysis steps depicted in Figure 1 can 
be automated, but as of now, the most difficult ones 
still requires analyst. As expected, the analyst has 
to write the harness for the target. He has to make 
ith compilable for all testing engines and he has to 
to control that automatic insertion of intrinsics does 
not break the program semantic.

While this research shows that automating most 
of the workflow is possible, combining both a fuzzer 
and DSE raises multiples issues that are yet to be 
addressed. Indeed, such heterogenous algorithms 
hardly work together. Experiments shows that 
fuzzing generates numerous test-cases that DSE 
replays significantly more slowly. It thus spends a 
significant amount of time performing its dry-run\cite{Corpus] to update its coverage with inputs received. The 
coverage synchronisation between engines is thus a 
bottleneck for symbolic execution.

Also, DSE in pure emulation requires a large 
number of syscall and external libraries modeling 
to scale on significantly larger code base. From the 
side-effect modeling perspective, scaling on signific-
antly larger codebase can be addressed using a 
concolic execution mode. Such an approach relies 
more heavily on concrete values during the execution 
which does not need to be modeled. Conversely, 
the reasoning power of the symbolic aspect is re-
duced as side-effects are not modeled symbolically.

Because of DSE limitations, current benchmarks 
results do not reflect a clear gain in combining 
fuzzing and DSE rather than running them separa-
ately.

6. Related work

Static analysis warning driven exploration Comb-
ing static analysis and dynamic testing to obtain 
better results than each technique taken separately

\cite{https://blog.quarkslab.com/remote-denial-of-service-on-cyclonetcp-cve-2021-26788.html} 
\cite{https://nvd.nist.gov/vuln/detail/CVE-2021-26788} 
\cite{Corpus replay to update the engine internal coverage. Inputs run are not mutated. The dry-run typically decides whether the input is worth being kept or not.
Table 4

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>D</th>
<th>V</th>
<th>Proto.</th>
<th>Function</th>
<th>Honggfuzz Cov</th>
<th>Honggfuzz Val</th>
<th>Triton Cov</th>
<th>Triton Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GB1</td>
<td>•</td>
<td>HTTP</td>
<td>httpParseRequestLine</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FMT</td>
<td>•</td>
<td>HTTP</td>
<td>httpSendErrorResponse</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IoF</td>
<td>•</td>
<td>HTTP</td>
<td>httpSendRedirectResponse</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BoF</td>
<td>•</td>
<td>HTTP</td>
<td>httpSendRedirectResponse</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>FMT</td>
<td>•</td>
<td>HTTP</td>
<td>httpReadRequestHeader</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>UaF</td>
<td>•</td>
<td>HTTP</td>
<td>httpSendRedirectResponse</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BoF</td>
<td>•</td>
<td>HTTP</td>
<td>httpParseContentTypeField</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BoF</td>
<td>•</td>
<td>HTTP</td>
<td>httpFormatResponseHeader</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>FMT</td>
<td>•</td>
<td>HTTP</td>
<td>httpParseContentTypeField</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>FMT</td>
<td>•</td>
<td>HTTP</td>
<td>httpParseContentTypeField</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>GB1</td>
<td>•</td>
<td>HTTP</td>
<td>httpDecodePercentEncoded</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>IoF</td>
<td>•</td>
<td>IPv4</td>
<td>ipv4ProcessPacket</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>SIGS</td>
<td>•</td>
<td>ARP</td>
<td>arpProcessReply</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>SIGS</td>
<td>•</td>
<td>ICMP</td>
<td>icmpProcessEchoRequest</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>BoF</td>
<td>•</td>
<td>ICMP</td>
<td>icmpSendErrorMessage</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>UaF</td>
<td>•</td>
<td>IPv4</td>
<td>ipv4FragmentDatagram</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>GB1</td>
<td>•</td>
<td>core</td>
<td>formatDate</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>SIGS</td>
<td>•</td>
<td>ETH</td>
<td>ethSendFrame</td>
<td>✓</td>
<td>-</td>
<td>×</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>UaF</td>
<td>•</td>
<td>IGMP</td>
<td>igmpProcessMessage</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>IoF</td>
<td>•</td>
<td>ICMP</td>
<td>icmpUpdateInStats</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

D: Default, V: Vulnerability, Cov: Covered, Val: Validated

has already been studied. From an error-condition inferred by a static checker Check 'n' Crash [30] aims at generating a test-case to validate if the error truly exists.

Another combination called Sante [31] uses the static analyzer Frama-C [32] to detect potential runtime errors. The result is combined with Pathcrawler [33], a DSE to generate a test-case and to confirm the alarms. DyTa [34] another utility, follows a similar approach.

Another category of related work rely on directed approaches. Gerasimov [35] uses static analysis warnings as targets for a directed DSE algorithm iteratively reducing the distance with the warnings to cover them. They use their own static analyzer Svace [36]. In another publication [37] they also study the reachability of the security warnings. The work of Li et al. [38] suggests an approach dedicated to Use-After-Free vulnerabilities where alloc and free primitives are used to drive the exploration. In a more general manner, multiple existing research works focus on directed approaches to cover specific locations of the program [14, 39, 40] but which are not necessarily driven by a SAST.

Fuzzing & Symbolic Execution combination

Various approaches combining these two testing techniques have been proposed in the past. Koushik Sen published in 2007 an Hybrid Concolic Testing approach combining the two [41]. Later, Driller [42] suggested a selective DSE algorithm launching Angr solely when the fuzzing is getting stuck. More recently QSym [43] intertwines the concolic execution within the fuzzing in a very light yet fast manner. Finally, multiple collaborative approaches allowing to combine heterogenous fuzzing engines have been proposed under the term ensemble fuzzing. Among them, we can highlight ClusterFuzz [44] by Google, EnFuzz [9], Deepstate [45], collabfuzz [46] or more recently OneFuzz [47] by Microsoft. To our knowledge none of these ensemble fuzzers uses a static analyzer as an input of test objectives.

7. Future work

These preliminary results open the way to further experiments and benchmarks. Multiple experiments can be made to optimize collaboration of test engines. We are working on improving the PASTIS framework by adding new fuzzing engines like AFL++ [17], adding slicing features to better
guide the exploration with more directed strategies or to enlarge the project scope to binary-only targets.

8. Conclusion

This paper summarizes what has been done as part of the PASTIS project and its implementation in the PASTIS framework. We depict a test suite enabling to discriminate and to choose a fuzzing and DSE engine for the PASTIS platform. We then describe the full workflow that we intend to automate. Namely, the paper discusses the process of analysing a source code with a SAST tool, how to embed this data in the final compiled program and how to automate the process of testing it with various heterogeneous testing engines. The result is a test corpus that can be integrated as tests in the project. An analyst, can use these results, to prune and ignore irrelevant alerts, performing the root-cause on crashes and focusing on the remaining alerts that have not been covered. This process is required in a wide range of industries like aerospace, automotive, defense, energy or any context that requires a higher level of insurance.

Acknowledgments

This research was realized by Quarkslab in the context of the PASTIS project financed by DGA-MI (Direction Générale de l’Armement, Maîtrise de l’Information).

References

[7] Perforce, Klocwork static code analysis for c, c++ and java, 2021. [site].
From Source Code to Crash Test-Case through Software Testing Automation


