Short Paper: Differential fuzzing of constant-time packages in JavaScript

Jeppe Fredsgaard Blaabjerg
jfblaa@cs.au.dk
Aarhus University

ABSTRACT

Timing-channels enable attackers to learn secret program data by observing the execution time of a program. Formally reasoning about timing-channels is difficult as accurately modelling the execution time involves correlating multiple program paths and requires extensive knowledge of source language, compilation steps, and hardware. Differential fuzzing promises an automatic way of discovering timing-channel following a general methodology that can be applied to programs written in any language. We test this promise by developing JsDifFuzz, a differential fuzzing tool for JavaScript. We use JsDifFuzz to evaluate JavaScript libraries for constant-time comparison, by generating input that maximises the wall-clock timing difference between executions. We evaluate examples from the literature and find that JsDifFuzz discovers timing-channels missed by other tools. We find that many of the libraries analysed are near constant-time for fixed-size input, but display timing-channels if input size is not known. We find that branch-prediction poses a challenge for using wall-clock time to guide the fuzzing.

1 INTRODUCTION

Timing-channels pose a significant security risk for a wide range of applications. The problem is exacerbated by the difficulty in formally reasoning about timing-channels, requiring highly detailed modelling of the runtime behaviour of programs. Nilizadeh et al. [5] introduce differential fuzzing, an automatic and general methodology for detecting side-channel in programs written in any language. The authors demonstrate the approach by an implementation that targets analysis of Java programs, however arguing that the methodology is general and could be used for programs written in any language. Their tool approximates the execution time of a program by counting byte-code instructions. The authors argue that this provides a good approximation of the actual execution time, not impacted by garbage collection and other processes running on the same machine. However, this simple modelling of time is not accurate in practice, where control-flow branches and cache behaviour influence execution time [4]. Furthermore, garbage collection can provide an exploitable side-channel [8] and is therefore beneficial to include in an analysis.

We test the claim that differential fuzzing can be applied to programs written in any language by developing JsDifFuzz, a differential fuzzing tool for JavaScript. JsDifFuzz uses high-precision timers high-precision timers available in Nodejs to measure the wall-clock execution time of programs. Like DifFuzz, we implement JsDifFuzz as a coverage- and timing-guided fuzzer. Coverage-guided fuzzing provides the fuzzing tool with information on how many lines of code, branches, or program paths were covered during execution. This feedback can be used when generating new input. Timing-guided fuzzing follows the same basic principle. We instrument JsDifFuzz with a scoring metric for the difference in execution time between runs and let new input be generated from the highest scoring inputs.

Differential fuzzing is particularly suitable for detecting when a program does not satisfy *non-interference.* Non-interference loosely is the condition that secret inputs do not affect public outputs. JsDifFuzz generates a public input *pub* and two secret inputs sec_1 and sec_2. We then analyse target program p that computes over public and secret input arguments by measuring the difference in time between executions using the two different secret inputs:

$$\underset{pub, sec_1, sec_2}{\text{maximize:}}\ \delta = |time(p(pub, sec_1)) - time(p(pub, sec_2))|.$$

We consider execution time, and thus timing difference, to be public output.

2 IMPLEMENTATION

We develop JsDifFuzz by heavily customising the JavaScript fuzzing tool Jsfuzz [3]. This enables us to utilise the existing infrastructure for coverage-guided fuzzing. As noted by Nilizadeh et al. [5], scoring the execution time of programs by measuring the wall-clock time makes the scoring unstable. To address this issue, we run target programs multiple times on each input and take the minimum score over all executions. Intuitively, an input may be assigned an artificially high a score if to garbage collection or other processes running on the same machine slow down one of the two runs. Thus, by taking the minimum over multiple executions we hope that at least one run using bad input will give an accurate score at which point the bad input may be discarded.

To improve input generation, we further modify JsDifFuzz to be an evolutionary fuzzer. Evolutionary fuzzers partition inputs into generations and uses the highest scoring input from generation k when creating the input for generation $k + 1$. This change significantly improves the performance of the fuzzer, by enabling information to be passed on from one generation to the next. We further explore the idea of evolutionary fuzzing by letting new input be generated from two *parent* inputs. If two high-scoring inputs a and b agree on input sizes or sequences in their data, we can exploit this agreement by using shared data as the basis for new input c, using random values where a and b disagree. The exact parameters for generation size and number of survivors from one generation to the next were heuristically chosen.

We let input be given by an array of JavaScript buffers and let the number of buffers needed be specified by wrappers for each target program. This allows buffers to have different sizes, a feature not present in DifFuzz, enabling us to investigates leaks through size. The benefit of this is shown the next section.

3 EVALUATION

We evaluate JsDifFuzz on a range of JavaScript packages for constant-time string and buffer comparison available via the Node Package Manager (NPM) [6]. We additionally analyse built-in comparison functions and examples analysed by DifFuzz [5]. Figure 1 shows our findings. Each test ran for 30 minutes on a MacBookPro14.3, Quad-Core Intel Core i7 2.8 GHz, 16 GB RAM, macOS v11.6. JsDifFuzz generated input consisting of three buffers and scored the input in accordance with the scheme outlined in Section 1. We perform two evaluations of each target program, one where the size of each buffer was less than or equal to 512 and one where the size of each buffer was 512. Columns 2 and 4 show the average difference in time between executions varying in their secret arguments for the final generation of inputs.

Benchmark	**Size** $\leq$ 512		**Size** = 512	
	Avg. δ	Std.Err.	Avg. δ	Std.Err.
Constant-time packages				
bcrypt	407.0	104.55	309.2	66.38
buffer-equal-constant-time	5065.2	18.19	353.9	133.13
compare-timing-safe	943.8	24.89	113.1	8.39
immutable-tuple	2.4	0.21	2.3	0.35
safe-compare	193.3	3.59	32.3	3.84
scmp	113.2	0.31	1.6	0.25
secure-compare-native	4.9	0.22	1.8	0.28
secure-compare	56.5	6.27	231.1	133.86
tsscmp	1434.8	11.91	91.2	5.91
tsse	653.5	1.31	13.6	0.85
Nodejs built-in				
Buffer.equals	54.9	0.30	6.7	0.55
crypto.timingSafeEquals	-	-	1.8	0.13
String.localeCompare	635.5	1.95	456.5	0.84
DifFuzz				
pwcheck_unsafe	943.8	24.89	5965.1	13.84
pwcheck_diffuzz	2039.4	31.51	200.5	17.67
jetty_fix_unsafe	5137.8	4.35	123.1	15.15

Figure 1: Measured difference in execution time in ns.

While many packages perform comparison in near constant-time for fixed-size input, we observe that some display significant timing-leaks when the size of input is not fixed, thereby leaking secrets. The package *tsscmp* [11] uses so-called *double HMAC*, hashing each argument before comparing them. However, the hashing function takes time proportional to the length of the input, hence leaking input size. Another package contains a comment arguing that *"buffer sizes should be well-known information"* [9]. Only the built-in *crypto* library of Nodejs alerts the user and raises an exception if given input of different sizes. For cryptographic protocols, constant-time comparison should arguably be performed on hashed strings, hence the size of input can be assumed to be fixed. However, we note that if solely usable in this setting, timing-channels can only leak the hashed secret. Unless the hashing function is broken, the original secret remains secure.

Nilizadeh et al. [5] use DifFuzz to analyse the three programs pwcheck_unsafe, pwcheck_diffuzz, and jetty_fix_unsafe using the byte-code instruction count as an estimate for execution time. Program pwcheck_unsafe checks that the lengths of the two inputs are the same, before iterating over the inputs in a simple for-loop, returning the result as soon as a mismatch is found. We note that timing-channel we find using variable size input is smaller than when using fixed-size input. For this program, a long execution time is obtained if the lengths of the inputs match and they agree on a long prefix. This suggests that our fuzzer finds a local maximum using relatively short input. Maximising the timing-channel requires increasing the length of the public argument, but doing so makes the program terminate early unless secret input is updated accordingly.

Nilizadeh et al. propose the program pwcheck_diffuzz (Listing 1) that is considered safe by their analysis.

```
1  let unused;
2  let matches = true;
3  for (let i = 0; i < pub.length; i++) {
4    if (i < sec.length) {
5     if (pub[i] !== sec[i]) {
6      matches = false;
7     } else {
8      unused = true;
9     }
10   } else {
11    matches = false;
12    unused = true;
13   }
14 }
15 return matches;
```

Listing 1: pwcheck_diffuzz

We observe that this supposedly safe program leaks the length of the input, a fact missed by DifFuzz. We also find that the program leaks the secret by the branching on line 5. It is well-known that control-flow branches are not constant time [2] due to branch prediction. While our tool fails to exploit this leak and generate input that maximises the timing difference, we observe that manually crafted input can reveal this side-channel. We expect a high number of mispredicted branches if the match is random. We test this hypothesis by constructing buffers pub, sec_1, sec_2 such that $pub = sec_1$ and $pub \overset{50\%}{\simeq} sec_2$, where $\overset{50\%}{\simeq}$ denotes that for all i, $P[pub[i] = sec_2[i]] = 0.5$. Measuring the timing difference between executions (pub, sec_1) and (pub, sec_2) we observe a δ of 2000, suggesting a timing-leak from branch-prediction.

4 CHALLENGES AND LIMITATIONS

Mitigating timing-channels in a high-level language such as JavaScript is difficult as optimising compilers may rewrite the source program thereby voiding any constant-time guarantees [10]. This problem is exacerbated by the just-in-time (JIT) compilation of JavaScript [1]. This makes it difficult to write truly constant-time code in JavaScript. As is considered best practice, the constant time comparison of Nodejs's crypto library is implemented in low-level assembly [7].

We find that there are a number of challenges remaining before fully realising differential fuzzing guided by wall-clock time. While the effects of garbage collection and other processes running on the same machine appear only minor in our findings, branch prediction poses a challenge for maximising timing side-channels.

As the branch-predictor becomes trained during testing, a highly-matching input can suddenly become fast, and a poorly-matching inputs slow, as the branch-predictor expects a match.

REFERENCES

[1] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. Jit leaks: inducing timing side channels through just-in-time compilation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1207–1222. IEEE, 2020.

[2] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 913–926, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385970. URL https://doi.org/10.1145/3385412.3385970.

[3] fuzzit.dev. Jsfuzz, 2021. URL https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/jsfuzz. Accessed: 2021-10-19.

[4] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1): 163–182, 2005.

[5] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 176–187. IEEE, 2019.

[6] npm Inc. npm, 2021. URL https://www.npmjs.com/. Accessed: 2021-10-19.

[7] OpenSSL. Openssl, 2021. URL https://github.com/openssl/openssl. Accessed: 2021-10-19.

[8] Mathias V Pedersen and Aslan Askarov. From trash to treasure: timing-sensitive garbage collection. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 693–709. IEEE, 2017.

[9] salesforce. buffer-equal-constant-time, 2013. URL https://github.com/salesforce/buffer-equal-constant-time. Accessed: 2021-10-19.

[10] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE, 2018.

[11] suryagh. tsscmp, 2018. URL https://github.com/suryagh/tsscmp. Accessed: 2021-10-19.