DuskFuzz: Encoding Side-Channel Information to Improve Blackbox Fuzzing

Ulysse Vincenti[1,2] [0009−0004−4432−9956] ✉, Thomas Hiscock[1] [0009−0001−5183−2291], and David Hely[2] [0000−0003−3249−7667]

[1] Univ. Grenoble Alpes, CEA, LETI, Campus, F-38054 Grenoble, France
firstname.name@cea.fr
[2] Univ Grenoble Alpes, Grenoble INP, LCIS, F-26000 Valence, France
firstname.name@lcis.grenoble-inp.fr

Abstract. Fuzzing is a widespread technique to identify system vulnerabilities. The process entails the generation of random inputs with the objective of triggering behaviors that were not previously anticipated. One strategy for significantly enhancing the capabilities of a fuzzer is to provide it with feedback on the execution. Unfortunately, the most effective feedback require binary access or software instrumentation and thus, can rarely be used on systems in production. Indeed, such systems are usually closed, both in terms of architecture, source-code and access to runtime information. This paper proposes a study on how time-series related to physical emanations produced by a device can be used as feedback for a fuzzer. It introduces DuskFuzz, a set of two new techniques for encoding time-related information in the interface medium of an AFL-type fuzzer. Fuzzing campaigns on an emulated device show that time-series of memory load/store events integrated with DuskFuzz improves coverage over a random fuzzer. In addition, through the analysis of electromagnetic emissions of an STM32F4, we show how to recreate coverage-related information. On a fuzzer aided by electromagnetic measurements, we observed branch coverage improvements up to 113% and 82% compared to a random fuzzer on a JPEG decoder application and zlib decompression. This work demonstrates that side-channel feedback offer a promising research direction to improve blackbox fuzzers.

Keywords: Side-Channel Analysis · Fuzzing · Hardware Security

1 Introduction

Fuzzing is a well-established technique for improving the security of systems. The wide adoption of fuzzing in the software industry, including open-source projects [12], is a testimonial to the effectiveness of this technique in practice. Thanks to active research, the range of fuzzing applications expands every day and now applies to Trusted Applications (TAs) and even hardware designs [33].

Fuzzing is the art of crafting unexpected inputs for a system under test in order to reveal unexpected behaviors. The latter are analyzed for discovering bugs or vulnerabilities. The effectiveness of a fuzzer is usually measured by how

much coverage (amount of code/behaviors explored) and bugs can be discovered in a given amount of time. Stressing a system only with random inputs (random fuzzing) is known to quickly reach a "plateau" in terms of coverage. To circumvent this limitation, fuzzers usually gather *feedback* on the system to determine whether an input was interesting. For example, the coverage-driven fuzzers in the AFL family [10] instrument the control-flow graph of the program (at compile-time or using binary instrumentation) to use the coverage as feedback. The availability of such feedback allows the use of advanced algorithms to explore the input space more effectively. However, most devices in production such as IoT devices, proprietary smartphones, automotive ECUs, secure elements, industrial systems, do not allow memory or debug access (for security reasons), meaning that little to no feedback are available on those devices. This paper investigates how fuzzing can be improved on such blackbox systems.

Side-channel analysis (SCA) is a set of techniques to recover secret information using physical activity of a device such as electromagnetic (EM) field or power consumption. SCA is generally used to exploit leakages from cryptographic implementations [16], but can also fingerprint the behavior of a system like executed instruction [11].

One key difference between side-channel information and logical information is that the side-channel information may be noised. This brings us to the main topic of this work: how can side-channel be used efficiently to generate feedback for fuzzing? Prior works [28,30,31,33] already investigated this question, showing that side-channel information may be used by a fuzzer.

In this paper, we revisit the use of side-channels for fuzzing on blackbox systems. We formalize a model that computes fuzzer feedback in two steps: first a side-channel analysis that builds a time-series of event counts and then an encoding of the time-series inside the fuzzer. The benefit of this approach is that it allows exploiting side-channel information that (1) can be obtained in an unsupervised fashion (i.e., without prior training on an open device) and (2) is fast to compute. The first constraint is mandatory to make the technique usable on blackbox systems and portable across devices. The second one is motivated by the fact that a fuzzer should maximize the number of inputs tested in a given amount of time. Through this work, we address the following research questions (**RQ**):

RQ1: How can we encode time-series information for a fuzzer to understand?
RQ2: Which metrics are relevant to extract from side-channel measurements to improve blackbox fuzzing?
RQ3: Which strategies can be employed in fuzzers to deal with the inherent noise is side-channel measurements?

In this context, our work makes the following contributions. We propose DuskFuzz an AFL-based fuzzer that incorporates two schemes for encoding time-series into AFL maps (section 4), as well as noise reduction strategies that can be applied in the fuzzer. We propose a rigorous benchmarking methodology based on coverage measurements to qualify side-channel based fuzzers.

Then, we evaluate the approach on two case studies targeting the STM32F4 microcontroller:

1. Simulation: we emulate the device to generate a memory load and store trace and evaluate the proposed encoding schemes. This first experiment allows benchmarking our solutions in a noiseless controlled environment (section 5).
2. Real world implementation: we evaluate our approach on an electro-magnetic side-channel analysis bench. Thanks to a leakage assessment, we show how to build a time-series from raw EM signals. Then, we evaluate the resulting fuzzer on multiple benchmarks (section 6).

The experimental results show that side-channel information, if correctly integrated in the fuzzer, can greatly improve the fuzzer coverage in some cases. In practice, our side-channel fuzzer surpasses a random fuzzer by 82% of branch coverage on the zlib decompression and by 113% on a JPEG decoder application.

2 Background

2.1 Assumptions

Fuzzing with side-channel feedback imposes more restrictions than traditional fuzzing. This section describes the assumptions on the security evaluator made in this work. First the target device is considered "blackbox", meaning we have no software access or debugging capabilities. We assume that the evaluator has access to a communication interface (e.g., serial communication) and physical access to the device. Thus, the evaluator can measure physical signals (EM field, power consumption) from the system. Synchronization of acquisitions would usually be required in real blackbox fuzzing scenario. This can be done by monitoring communications or triggering on specific patterns found side-channel measurements. Since many side-channel synchronization solutions exist in the literature [3], we consider this subject to be out of the scope of this paper. Therefore, in this work, the device under test emits a trigger signal that allows the side-channel traces to be synchronized.

Embedded devices often use peripherals, these interactions might introduce additional noise in the side-channel trace. This work considers only deterministic stateless target devices. A stateful device is generally handled through a dedicated fuzzing engine [26]. The feedback remain the same as a stateless fuzzer. Thus, this work could also be integrated in a stateful fuzzer.

2.2 Overview of Coverage-driven Greybox Fuzzing

The goal of this work is to investigate whether fuzzers, more precisely greybox coverage-driven fuzzers, can benefit from side-channel information as an alternative to traditional instrumentation. In this section, we remind the essential concepts about those fuzzers required to understand this work.

A coverage-driven fuzzer optimizes its input selection to maximize the code coverage of the target software. The rationale is that increasing coverage increases chances to explore unexpected paths and hence trigger bugs. AFL-like fuzzers are arguably the most common coverage-driven fuzzers. They instrument the software at compile-time (or runtime using dynamic binary translation or dynamic binary instrumentation [10,20]) so that each time an edge is taken in the Control Flow Graph (CFG), a counter associated with this edge is incremented.

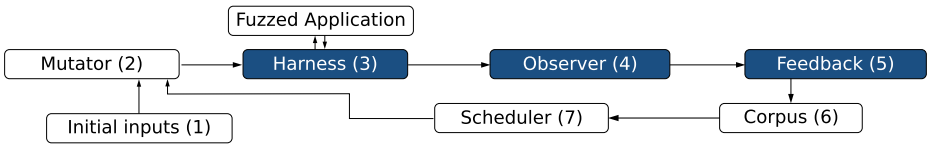


Fig. 1: A simplified pipelined vision of a fuzzer that uses LibAFL framework.

The abstraction of a fuzzer considered by LibAFL is depicted on Figure 1. The fuzzing starts with a set of initial corpus of inputs (1) and perform some random mutations on it, like byte insertion, bit swap (2)... Then, each input is passed to a harness (3). It serves as an interface between the fuzzer and the fuzzed application. This abstraction is quite handy in the context of this work, where we can delegate the input execution to an external device at this stage. During the execution of the target, there are one or more "Observers" (4) whose role are to collect multiple metrics (standard output, exceptions, hit-count map...). The Feedback stage (5) evaluates the novelty of an input from those observations, those considered interesting are added to the Corpus (6). The Scheduler (7) will then choose the most promising elements in the Corpus (6) and start a new fuzzing loop.

AFL maps are matrices where each edge of the control-flow graph (CFG) met during the execution is registered. They enable the detection of novelty when a new input is executed. During the execution, when a new basic block ($BB_{current}$) is met, AFL calculates the edge identifier as: $E_j = (@BB_{prev} >> 1) \oplus @BB_{current}$, where $@BB_{prev}$, $@BB_{current}$ are respectively the addresses of the previous and current basic blocks.

The edge identifier is then used as an index of the AFL map, to be incremented. For example, if the input "AA" triggered edges: E_0, E_1, E_0, E_0, E_3 the map will look like this: $M_a = [3, 1, 0, 1]$. During a second execution with the input "BB", that triggered edges: E_0, E_1, E_2, E_2, E_3, the map will be: $M_b = [1, 1, 2, 1]$.

A quantization process is then performed to reduce the range of values in the map. It prevents the detection of novelty on repetitive loop sequences and allows working with an 8-position bitmap. AFL typically uses 9 buckets: [0, 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+]. Each bucket of this mapping is transformed into a power of 2, except for the value 0 which is kept intact. Therefore, M_a becomes

$\hat{M_a}$ in this new map, where each value is encoded as a power of 2, consequently, a single bit is set.

The novelty detection is performed simply by comparing a history map M_H with each of the new maps. A history map, will evolve through the fuzzing process, it is therefore indexed through time, with n the number of input evaluated. The n-th input "XX" with an associated map $\hat{M_n}$ is considered interesting if any bit with value '1' of $\hat{M_n}$ is absent of $M_H(n)$. The bitwise OR between $M_H(n)$ and $\hat{M_n}$, generates the union of the two maps. XORing this map with the new map $\hat{M_n}$ allows us to get the bits that were present in the new map but not in the history map. Verifying that this resulting map has some bits to 1, allows us to identify that new behavior were observed. More formally, if $(M_H(n)|\hat{M_a}) \oplus M_H(n) > 0$, then the input is considered interesting.

The history map is then updated with the newly discovered bits. The history map $M_H(n+1)$ takes as value the joint map of the history map and the input, as said before, the bitwise OR of its old values $M_H(n)$ and the new input $\hat{M_a}$. In other words, the state $n+1$ of the history map is calculated: $M_H(n+1) = M_H(n)|\hat{M_a}$.

3 Related Work

Some solutions exist to fuzz blackbox devices without using side-channel information. Debug interfaces, if available allow tracking the control-flow at runtime [8, 24]. Other works [4, 27], analyze interactions of smartphone apps with external physical devices (e.g., a smart bulb) to craft packets and fuzz the connected equipement. An alternative is to monitor communications to forge new packets [18]. Finally, another well-known approach is to employ grammar-based fuzzing to improve input creation as done on network fuzzers [9, 15, 21].

Several works studied how to combine SCA with fuzzing. Sperl et al. [31] describe the construction of a power side-channel aided fuzzer on a STM32F4. Using a profiled target, they built a branch detection classifier as well as a branch destination classifier. Then, they described an algorithm for reconstructing the control flow graph from a side-channel trace, the latter can be used to recompute CFG edges. They did not implement their technique in a real fuzzer, so the real performances of their approach are hard to estimate. However, this technique relies on profiling, which makes it work in a blackbox fuzzing context.

A presentation made by Riscure at hardwear.io conference [28] described some ongoing work on side-channel aided fuzzing. Choosing a custom smartcard firmware as a target, they managed to use power consumption to recreate coverage. The traces are processed using an advanced pipeline of dimension reduction and clustering to label them and perform novelty discovery. Using the power consumption, they achieved a 28% coverage gain over a random fuzzer on the smart-card fuzzing, trying to guess some secret applet commands. Another experiment used timing leakages on the U-Boot of a Pinephone and recovered 16 commands in 2 hours without any starting corpus. EM emissions and power consumption were not used on this benchmark because of the noisiness of the

target. This presentation lacks from enough details to allow for reproducibility or comparability.

Su et al. [33] introduced a fuzzing method for FPGA hardware IPs based on power measurements, using K-means clustering to identify interesting inputs. They compare their solution against a random fuzzer and a no-feedback fuzzer, a random fuzzer samples random inputs from scratch each time, while the no-feedback fuzzer takes an always-true feedback, and therefore, will consider interesting any input mutated. Depending on the benchmarks their solution scored better or equally than the random fuzzer, but always worse than a greybox fuzzer.

Sesterhenn [30] proposes a fuzzing method for STM32F0 that uses power side-channel analysis, comparing new input traces to a corpus via distance metrics. It effectively identifies interesting inputs in an insecure password check and HTTP parser, but lacks comparisons with other fuzzers, making its true effectiveness unclear. Barredo et al. [1] enhanced greybox fuzzer seed selection by incorporating side-channel observations. They leverage anomaly detection algorithms on Raspberry Pi 3b EM traces to detect outlier in memory-accesses. AFL++'s initial corpus is filtered to only retain inputs that generate side-channel outlier. The framework showed unique crash discovery improvements against standard AFL++ fuzzing.

Several works investigated the use of performance related metrics as feedback for their fuzzer. The wall-clock timing was used by Chen et. al [5], others, like SlowFuzz [25] and DiffFuzz [23] exploit the number of instructions to reduce the noise. Basu et al. [2] used the number of cache misses and unique cache sets accessed during the fuzzing process.

As mentioned previously, one of the main objective of this paper is to consider simple and fast metrics, and encoding to get coverage information from side-channel. Sesterhenn, Su et. al and Riscure [28, 30, 33] perform clustering on the entire power traces acquired, this technique is suboptimal regarding performance. We propose an alternative using a way to encode the side-channel information into AFL maps. A lack of comparability between the different SCA-based fuzzing solution has been spotted, we believe that this paper is a first step towards the creation of a robust methodology for future contributions.

4 DuskFuzz: Encoding Time-series into AFL Map

Side-channel information can be of different kinds. Its diversity ranges from a single integer to an array of floating points or even time-frequency data. The granularity of the information depends on the leakage sources and processing methods used. Our idea here is to abstract the side-channel analysis processing (which is device-specific) with a time-series. This makes our fuzzer applicable to physical side-channels (emissions, power-consumption, sound) or any source that fits in the time-series paradigm: sampling of performance counters, or micro-architectural events.

A time-series is a record of quantity through time. It can be represented as an array of tuples (timestamp, quantity). For example, the time-series studying a moving object at constant speed could be: [(t_0, x_0), (t_1, x_1), (t_2, x_2)...] = [(0 s, 0 m), (1 s, 9 m), (1.5 s, 13.5 m) ...]. We consider time-series where t only grows positively, therefore, $t_{n-1} < t_n$.

A typical physical side-channel acquisition done with an oscilloscope will output a time-series of equally-spaced digitalized samples. Then, an analysis process will take as input the side-channel trace and output a time-series of "relevant" information. This process can be achieved using simple metrics (e.g., windowing, threshold comparison) or more complex ones such as machine-learning based classifiers.

Remember from section 2 that the AFL map is an array of N bytes that generally holds counts of events. Thus, it is important to design the side-channel processing to generate meaningful events.

We propose 2 techniques to encode time-series into the AFL-Map, T and ΔT. They form DuskFuzz, an AFL type fuzzer that incorporates side-channel feedbacks.

4.1 Spill the T

To T encode the time-series, we simply have to accumulate the quantities using the raw timestamps as index in the resulting AFL-Map. In the case of timestamp larger than the AFL map size, the remainder of the Euclidean division (modulo) with the AFL map size is used as index.

To perform such operation, wrapping and accumulating the time-series on itself is the simplest solution. For example, an input time-series: [(0, 1), (1, 3), (5, 4), (10, 35)] will be encoded into a map of 5 elements to: [40, 3, 0, 0, 0].

The T encoding algorithm is presented in Algorithm 1. For each element in the time-series given as input, the algorithm increments the output matrix at the time index modulo the size of the output matrix. The intuition behind this encoding (presented in Algorithm 1) is that it keeps the absolute time integrity of the data in the resulting AFL map. This is interesting because it increases the likelihood of differentiating behaviors. For example, if the input executed triggered the execution of a new function, the whole time-series would be shifted to the right, resulting in a totally different AFL map. However, by construction, it is highly time-dependent and is subject to jitter which is frequent in side-channel measurements.

4.2 Spill the ΔT

Another encoding that has been explored in our research is the ΔT event encoding. In the version shown in Algorithm 2, for each consecutive elements of the time-series, the time difference between those events is used as index to increment the output matrix by the quantity of the latest event of the 2.

The intuition here is to use relative time instead of absolute time. By using the relative time, we make sure that if there is jitter in the time-series, only a

few cells of the map will be impacted. This solution is still prone to a small error propagation, if an erroneous point j is present in the time-series, the AFL map will encode two errors: first, the time difference between the event $j-1$ and the erroneous event j; and then, the time difference between erroneous event j and the next correct event $j+1$.

The use of relative time allows detecting pattern such as `for` loops. If a repetitive behavior A is seen n times at the execution of input I_i and the execution of input I_{i+1} triggers $n+1$ times behavior A, the $(n+1)$-th behavior will be placed in the same map's cell as the n-th. The encoding of the following behaviors in the time-series won't depend on the previously encoded behavior A.

On the other hand, in the case of T encoding, if a repetitive behavior is performed more times than previously, the use of raw timestamp to encode the information will make the full encoding shift to the right (as previously cited). Any repetition of a behavior A will generate a new map and be considered as interesting.

A special case of the input time-series are binarized time-series. In this case, the time-series are simple records of events. Therefore, it can be simplified from $[(t_0,\ x_0),\ (t_1,\ x_1),\ (t_2,\ x_2)...]$ to $[t_0,\ t_1,\ t_2...]$ because each of the x_n are set to 1. Such time-series appear when the side-channel traces have been preprocessed to detect the presence of events. T and ΔT work similarly, they will increment the value in the AFL map instead of accumulating the value associated with the timestamp.

During the accumulation in the AFL map, a wrap is performed if a cell value overflows the support on which it is stored. In the classical AFL, NeverZero [10] prevents the value from going back to 0 during a wrap. This technology was not implemented in our technique. However, we used a map of `uint16_t` instead of `uint8_t` to prevent the wrap from happening. This task was performed by doubling the AFL map size inside of AFL and mapping each `uint16_t` onto 2 `uint8_t`.

4.3 Mitigating Noise in Feedback

Side-channel measurements and the time-series derived from them are inherently noisy. Coverage-driven fuzzers such as LibAFL are not designed to handle noisy observations.

This implies that to use LibAFL with side-channel feedback, the feedback must have as low noise as possible. One of the solutions addressed here is to use internal AFL tools as noise reduction tools. The bucketization process, as presented in subsection 2.2 can also be used to reduce the noise. By dividing the number of possible values, the light fluctuations caused by noise will have fewer chances to trigger the detection of new behavior.

Another noise reduction can be applied before LibAFL's internal bucketization by simply dividing the values in the AFL map.

4.4 Summary

In this part, we first defined time-series and then explained their use in fuzzing with side-channel. We described DuskFuzz, which enables the encoding of side-channel information as time-series as an AFL map with two different techniques: T & ΔT (**RQ1**). Finally, we depicted how quantization is used for noise reduction, this also presented some initial response to **RQ3**. The next sections evaluate the quality of those encoding on some case studies.

5 Case study 1: Load/Store Memory Traces in Emulation

To evaluate the quality of the proposed encodings, we generated time-series related to micro-architectural events using an emulation framework. We emulated a STM32F4 target because it is the target used in the real-life evaluation section 6. The choice of this target was motivated by a wide usage in side-channel academic works[3] and embedded devices.

It is well-known in the side-channel community that memory activity, especially in FLASH generates lots of leakages on power consumption and EM measurements. This was confirmed in the leakage assessment conducted in subsection 6.1. Thus, we decided to generate a binary time-series that represent the memory load and stores to validate the encoding schemes proposed. By integrating the new encodings in a LibAFL fuzzer, we compared our solutions to 3 other fuzzers on 3 different benchmarks.

5.1 Load & Store as a Fuzzer Feedback

To obtain accurate predictions of memory loads and stores, we built an emulation framework of the target device, depicted on Figure 2. Our modified LibAFL fuzzer interacts with the emulator thanks to a custom Executor. As shown on Figure 2, we chose a Unicorn based emulation of the STM32F4 to execute inputs and collect feedback. Unicorn is a front-end for QEMU, a CPU emulator that uses dynamic binary translation. Unicorn allows defining hooks on many events that occur during the emulation: basic-block transition, memory access, instructions execution...

Vanilla Unicorn cannot run a STM32F4 firmware since it only emulates the Cortex-M4 processor. To make the emulation work, one possible approach would be to emulate, or stub, all the peripherals of the STM32F4 used by the firmware. Luckily, there is a much simpler approach: create a snapshot of the target at the fuzzing entry point. Since the fuzzed part of the software does not use peripherals, we do not need to emulate them. During the emulator initialization, the different memory regions (SRAM, FLASH) and the registers are restored to the snapshot values. The memory address of the input buffer was carefully recorded so that

[3] In July 2025, We found 309 results on Google Scholar for papers that used STM32F4 on the topic of side-channel analysis. The request on Google Scholar was *"stm32f4" AND "side-channel"*.

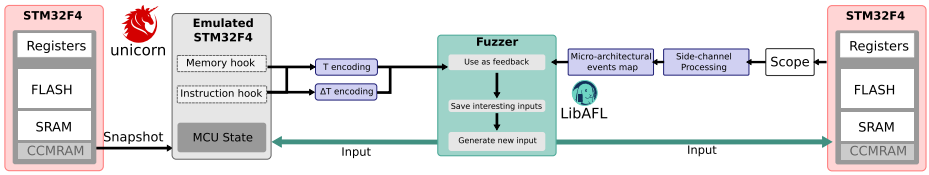


Fig. 2: Framework implemented.

the fuzzer could modify its contents with each execution. Finally, the program counter is set to start back from the breakpoint position. This whole procedure allows re-creating an emulated device from which we can implement hooks for monitoring micro-architectural events.

Using a hook invoked at each instruction, the emulator can generate an emulation tick which holds the current emulation cycle. Since not all instructions execute in a single cycle on the Cortex-M4, this modeling is not cycle-accurate. This emulation cycle can then be used to index emulated micro-architectural events in a time-series. Finally, hooking the memory read and writes makes it possible to recreate a time-series of memory events.

5.2 Fuzzers to be Compared

We compare the SCA based fuzzers against 3 types of fuzzers:

- A state-of-the art greybox LibAFL fuzzer, that uses CFG edges coverage feedback. Since this fuzzer requires software instrumentation, we recompile a native version (x86_64) of the firmware with `LLVM pc-guard`.
- A no-feedback fuzzer that will find every input executed interesting, and therefore, will mutate every inputs without preference.
- A random fuzzer that will generate completely random inputs without mutation at each execution.

The first type of fuzzer gives a hint on the best coverage that can be reached in a given number of executions. The second and third gives an estimation of the coverage reached by a complete blackbox attacker. The use of two types of random fuzzers was inspired by Su et al. work [33]. Those three fuzzers are essential to consider before drawing conclusions on the effectiveness of side-channel feedback.

Our side-channel aided fuzzer is declined into 2 variants, both having the same information source, memory read and write. They differ in the encoding of the information, the first one uses the ΔT encoding while the other uses the T encoding.

In order to estimate code coverage for each benchmark, we used the `gcov` tool. At the end of the fuzzing campaign, the corpus obtained on each fuzzer is replayed on a version of the firmware recompiled with coverage instrumentation using `gcc`. The number of branches reached by each fuzzer is compared to the

random fuzzer, which serves as a baseline. The experiments are repeated five times and the results averaged to reduce the non-determinism of the fuzzing process.

5.3 Benchmarks

We compare the fuzzers against 3 benchmarks: cJSON parser, libjpeg-v8 parser (from libjpeg-turbo) and zlib decompression algorithm. libjpeg-turbo and zlib are part of Google's FuzzBench [22] benchmark suite. Because this framework is designed for native applications, only few benchmarks can actually fit on a microcontroller such as the STM32F4. FuzzBench is a fuzzing benchmark created by Google to test multiple fuzzers in similar conditions. It offers multiple targets and has been pledged by the academic community.

The firmware used during emulation (this section) and the practical experiments (section 6) are the same. The compiler options are set to build a "release" version of the firmware with `gcc -O3` optimizations.

5.4 Fuzzing Parameters

Each fuzzer is parameterized to perform a search of at most 1,000,000 inputs with a 1000 bytes maximum size. The fuzzing always starts with the same random initial corpus of 100 inputs, sampled uniformly, of sizes between 100B and 1000B.

The limit on the number of executions is here because we are comparing fuzzers with different types of instrumentation. In one case, the fuzzer studies an execution with an `LLVM pc-guard` instrumentation and in the other case an execution performed on an emulated device with an overhead. This means that the fuzzer with simple compiler instrumentation would have many more opportunities to find coverage than the fuzzer on an emulated device would have in the same amount of time.

5.5 Results

In Table 1, we compare the coverage results of our emulated fuzzing campaigns and the 3 other reference fuzzers presented in subsection 5.2.

The first line of the table presents the no-feedback fuzzer's results, filled by 100% branch coverage, because it is used as a reference. Therefore, 100% does not mean a 100% branch coverage of the benchmark, but 100% of the no-feedback fuzzer performance on branch coverage. The second line also presents a blackbox fuzzer results, the random fuzzer's results.

As expected, the highest coverage ratios are reached by the native grey-box fuzzer (`LLVM pc-guard`). On average, it gets better coverage than the no-feedback fuzzer by 81.75%.

The ΔT encoding gives on average 35.81% better results than the no-feedback fuzzer and better results than the T encoding by 4.52%. However, for both the T and ΔT encoding, on the cJSON benchmark, the fuzzers fall between the

Table 1: Comparisons between random, greybox and emulated feedback mean and standard error results on 3 different benchmarks (no-feedback as reference for 100% branch coverage).

Fuzzer	**Feedback**	**Benchmark**		
		cJSON (%)	libjpeg-v8 (%)	zlib (%)
No-Feedback	None	100.0 ± 0.51	100.0 ± 1.97	100.0 ± 1.56
Random	None	112.15 ± 0.33	101.39 ± 2.54	90.78 ± 5.76
AFL-Map fuzzer	`LLVM pc-guard`	142.87 ± 1.49	263.08 ± 1.54	139.31 ± 1.06
	T encoding	105.14 ± 3.05	155.97 ± 3.46	132.76 ± 1.66
	ΔT encoding	108.87 ± 1.27	165.16 ±2.93	133.41 ± 1.01

no-feedback and the random fuzzer results. As discussed in subsection 6.6, the reason is that the feedback (memory events) does not contain enough information about this specific benchmark to achieve good coverage.

The results show that the emulated timestamped memory events information is relevant to the fuzzer. The improvements observed are heavily influenced by the benchmark used.

5.6 Summary

Thanks to the STM32F4 emulation, it was demonstrated that time-series encoding as an AFL map is a good candidate for recreating partial coverage information.

It can even help a side-channel based fuzzer to outperform a random fuzzer. However, those results model a noiseless side-channel observation of the memory activity. It is therefore important to validate these results on a hardware target.

6 Case study 2: Practical Blackbox Fuzzing on STM32F4

The emulation of the target allowed for the theoretical validation of the T and ΔT encodings with promising results. However, they have to be tested on a real target. This section describes the practical implementation of a side-channel aided fuzzer on the STM32F4. We rely on the experimental setup described in subsection 6.1 and Figure 2. The target firmware is the same as the one used in emulation section 5. It is compiled in release mode (-O3 optimizations) and the microcontroller is running at 168 MHz. This setup reflects the configuration of a device in production.

A trigger managed by the software surrounds the target functions to simplify side-channel measurements.

6.1 Investigation of Side-channel Leakages

Prior to building time-series related to the micro-architectural events during an execution or in other terms a side-channel feedback for fuzzing, a study must be made about the kind of information that can be extracted on the target.

The target selected for this work is an STM32F4 device, available on the STM32F4DISCOVERY development kit.

Multiple source of side-channel information are available on a physical target device. Power consumption and EM emissions are the most common for SCA because they provide good temporal resolution regarding the operations of the target [16]. While power measurements could be of interest and allow for a reduced sampling frequency, they are harder to employ on a blackbox system: knowledge of the PCB is a prerequisite, some soldering is needed, and the resulting signal encompasses lots of noise from other components of the system. On the opposite, EM measurements are not intrusive and have a good temporal and spacial resolution, the only real difficulty is finding a good probe location. For those reasons, our study favors EM field side-channel measurements.

Our acquisition setup consists of a near-field Langer probe (RF-B 0.3-3) mounted on an XYZ motorized table, the raw signals are amplified using a +30dB amplifier. Unless specified, side-channel traces are acquired using a Tektronix MSO oscilloscope with 2.5GHz banding and a sample rate of 5GS/s.

First Observations While running a JSON decoder (cJSON implementation) and moving the probe manually, we quickly found a probe location where a clear signature appears on the EM traces (see Figure 8). By zooming, it seems that the traces are made of repetitive patterns, such as the one shown between the vertical red lines on Figure 8.

Prior works showed that a wide variety of micro-architectural events can be witnessed through EM leakages [6,14,16,19]. By the time those traces were made, we had a strong suspicion that those patterns were related to some memory activity, since this was already reported on ARM devices by previous work [19].

6.2 Characterization of Memory Activity

Data Load Activity We designed a simple firmware on the target device that takes as input a number (N) of loads to perform on a fixed 40×32 table stored in FLASH. The firmware builds a payload in SRAM that contains N 32-bit load instructions (i.e., `LDR` in assembly). The following elements are randomized:

- The position of load instructions in the payload. Initially, the payload is made of 200 `NOP` instructions, the `LDR` instructions can be placed at any of the 200 slots available.
- The index in the table is chosen randomly for each load instruction emitted.

In addition, a trigger signal managed by the firmware surrounds the start and end of the payload for synchronizing acquisitions.

On Figure 3, we show some traces obtained while running the characterization firmware. With 1 load performed, all traces show a single pattern (placed differently because of the randomization). However, when varying the number of loads it can be observed that the number of patterns does not match the number of loads. For example, with 4 loads (middle graph), we only observe 2 blue patterns.

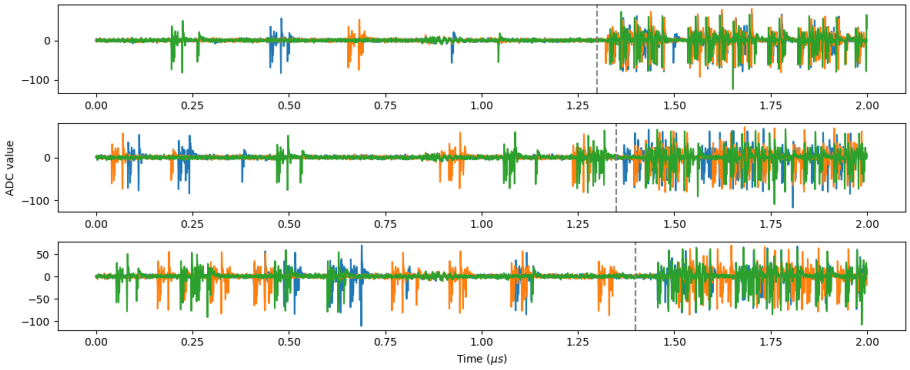


Fig. 3: EM traces of the load characterization, with different number of loads. Top: 1 load, middle: 4 loads, bottom: 10 loads. The grey dotted lines roughly indicates the end of the payload.

Those observations strongly suggest that the leakages are linked to some sort of cache inside FLASH memory. Indeed, the documentation of the STM32F4 specifies that the microcontrollers are equipped with instruction and data caches inside the FLASH memory [32]. The FLASH data cache has a fully-associative structure of 8 lines of 128 bits (i.e., 4×32 bits words).

To confirm this hypothesis, we can try to correlate the number of patterns with the theoretical number of misses. The latter is obtained by collecting the addresses accessed by the firmware and with a simple simulation of a fully-associative 8 cache lines.

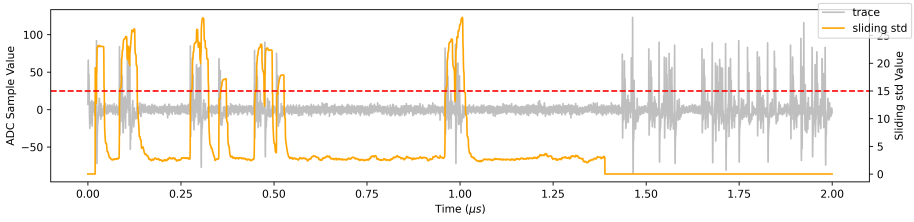


Fig. 4: Metric retained detecting memory activity: a sliding standard deviation.

We computed Pearson's correlation with different metrics. The first one is a pattern matching using Euclidean distance, we apply a threshold of 100 and count the number of elements below this threshold. The second one is a windowed standard deviation (sliding across the trace) on which we count the number of elements greater than 15. We represent on Figure 4 a trace with sliding standard deviation of 100 samples. The metric is forced to 0 outside of the trigger area. The results shown on Table 2 demonstrate that the number of misses is better related to the number of patterns than the number of loads. Currently, the sliding standard deviation is the best metric we found, reaching a correlation of 93% with the predicted number of misses.

Table 2: Results of correlation experiments.

Metric	**Pearson's Correlation**
Number of loads vs. pattern matching	0.75
Number of misses vs. pattern matching	0.82
Number of loads vs. sliding std	0.86
Number of misses vs. sliding std	0.93

This metric works because memory activation generates activity which translates into a high local variance of the EM signal. Any activity detection metric (e.g., energy, standard deviation, integral), could be relevant to measure the memory activity. More advanced machine learning models such as neural networks may also outperform the sliding standard deviation metric. However, the latter has two essential features considering fuzzing: it is easy and fast to compute, and it requires no prior training. Here, we instrumented the target to understand the underlying mechanisms, but the metrics proposed can be prototyped directly on a blackbox device, using traces from random inputs (e.g., Figure 8).

Instruction Load Activity The reference manual of the STM32F4 also states there is an instruction cache in the FLASH [32]. Since FLASH data cache leakages are observable, it is legitimate to ask if the FLASH instruction cache leaks too. To answer this question, we designed a firmware that performs a conditional branch at the middle of a sequence of `NOP` instructions. Then, we collect two group of traces: one with the branch taken, the other one the branch not taken. We performed a cartography above the chip to select a probe position. At each position, we collect 10,000 traces spread equivalently between the two groups and compute Welch's T-Test [29].

The results at the best probe position (i.e., the highest t-values observed) are shown on Figure 5. First, we can see that periodic patterns similar to those attributed to cache load misses (e.g, see the grey area Figure 5) appear on the traces. The interval between the two patterns is of 238 samples (47.6ns), as shown

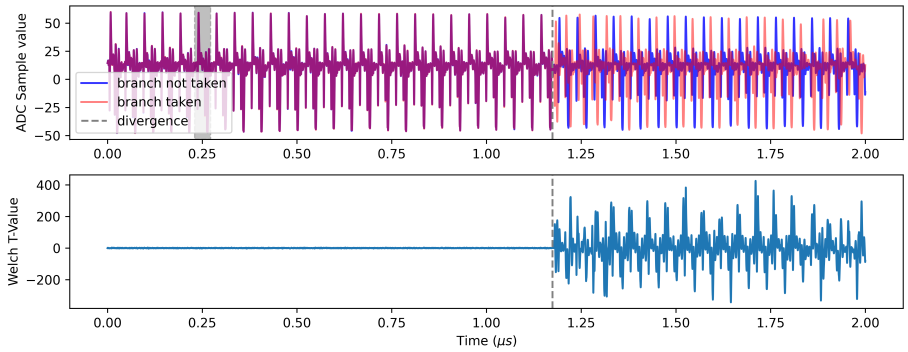


Fig. 5: Analysis of a conditional branch. Average traces for each case (top), Welch's T-Test value (bottom).

on Figure 6. Interestingly, it corresponds to 7.998 cycles at 168 MHz. Since, we are executing a sequence of Thumb `NOP` instructions, encoded on 16 bits, there are 8 `NOP` within an instruction cache line. Thus, our interpretation is that each pattern on Figure 5 is generated by the prefetch of the next instruction cache line. On Figure 6, we can see the traces of delay between peaks as filled lines and the source traces with lower opacity. When a branch is taken, it can be observed on Figure 6 that the distance between peaks decreases. Our interpretation is that the processor is pre-fetching cache lines faster until it has enough instructions loaded ahead.

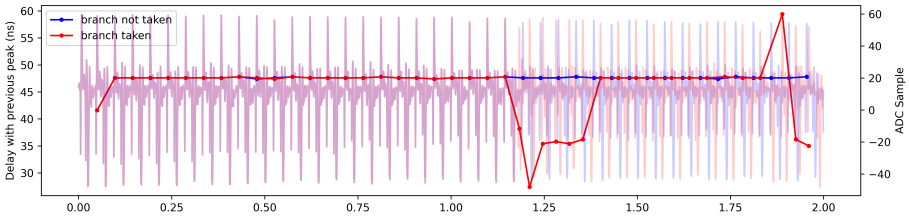


Fig. 6: Study of distances between peaks.

The takeaway from this is that very simple non-profiled metrics (e.g., windowed standard deviation) allow detection of cache misses in instruction or data FLASH caches and answers **RQ2**. A deeper analysis and interpretation of the leakages is left as a future work, since those leakages are already interesting to consider for fuzzing. The logical cache miss information obtained through SCA is used in future sections as fuzzing feedback.

6.3 Noise Reduction with Majority Voting

In subsection 4.3, we introduced how our technique used internal AFL quantization tools to reduce noise. The second technique used is to identify the most frequent value in each cell of multiple side-channel observations of an input. This is done using a majority voting on a repetition of AFL map of the same input. To perform such task, for each input, we acquire 5 similar traces and compute their corresponding map. The 5 maps are then merged using a statistical mode on the same cells of the different maps, this operation corresponds to a majority voting on each cell. This second element of response finalizes our answer to **RQ3**.

6.4 Side-channel Feedback

The first step of the side-channel processing implemented in the fuzzer, is to extract the region of interest, where the target function is executed, from the trace. The region of interest is extracted using the trigger signal which is acquired in parallel of the EM traces. This allows us to reduce the processing time and to only feed the fuzzer with information about the actual processing of interest.

We use the sliding standard deviation metric explained in subsection 6.2 to retrieve the memory activity information. The resulting trace is then dealt with separately for the T and ΔT encoding.

***T* Encoding** The sliding standard deviation trace is applied with a window size of W samples and an overlap of O samples. For each window, we count the number of points above a certain threshold and use it as the window's score. Each score is considered as a timestamped event (the timestamp being the index of the window), they are stored into a map as explained in section 5. Finally, we compute the statistical mode of 5 similar observations of the map to reduce the noise as described just before.

***ΔT* Encoding** The sliding standard deviation trace in this case is binarized using a threshold. The trace is then analyzed so that we have a resulting list of the indices of the rising edges of the binarized trace. The distance between each rising edge is then used to construct the ΔT as explained in section 4. Finally, just as for the T encoding, we perform a majority voting on the 5 similar maps.

Parameters The parameters retained in our experiments are a window size $W = 300$ with an overlap $O = 20$ samples. They are derived from the leakage assessment study in subsection 6.2. For each benchmark, the evaluator must fine-tune the standard deviation threshold. This is done during the initial corpus evaluation. If the corpus grows too fast, the fuzzer is most likely observing different feedbacks for the same input, therefore, the threshold needs to be tightened. On the other hand, if the discovery speed is too low, it might not differentiate feedbacks from different inputs, therefore, the threshold needs to be loosened. The other parameters can be optimized in the same manner.

6.5 Evaluation

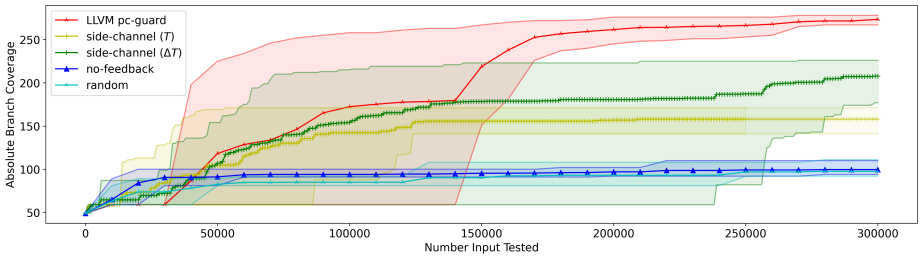
Figure 7 shows the results of fuzzing campaigns using real EM side-channel feedback. The average results are shown by solid lines. The minimum and maximum of the results distributions are shown as colored zones. We analyze in Table 3 each corpus at the maximal number of inputs reached by the side-channel fuzzer (see Figure 7). The results are given relative to the no-feedback fuzzer, as in subsection 5.5.

Once more, the results are given based on the number of inputs tested, not time spent on the target. This is because a fair comparison would not be possible, as the execution time is different between the greybox and the blackbox fuzzer. The goal of this work is to validate the quality of the side-channel information to create coverage, not the performances which will be improved in future engineering work.

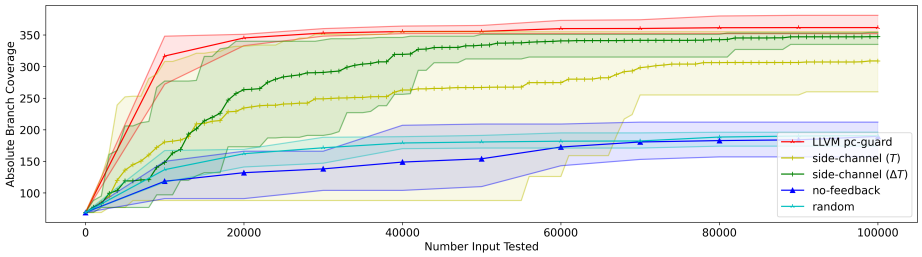
Table 3: Comparisons between 3 (random, greybox and real side-channel) mean and standard error fuzzers results on 3 benchmarks, taking no-feedback (at the same number of input tested) as reference for 100% branch coverage.

Fuzzer	**Feedback**	**Benchmark**		
		cJSON (%)	libjpeg (%)	zlib (%)
No-Feedback	None	100.0 ± 1.20	100.0 ± 2.44	100.0 ± 4.88
Random	None	111.13 ± 1.84	97.99 ± 3.30	100.74 ± 2.02
AFL-Map fuzzer	`LLVM pc-guard`	117.45 ± 3.9	274.5 ± 1.88	190.92 ± 2.51
	T encoding	88.12 ± 1.14	158.63 ± 3.80	163.06 ± 6.35
	ΔT encoding	105.06 ± 1.47	208.63 ± 7.60	183.31 ± 1.51

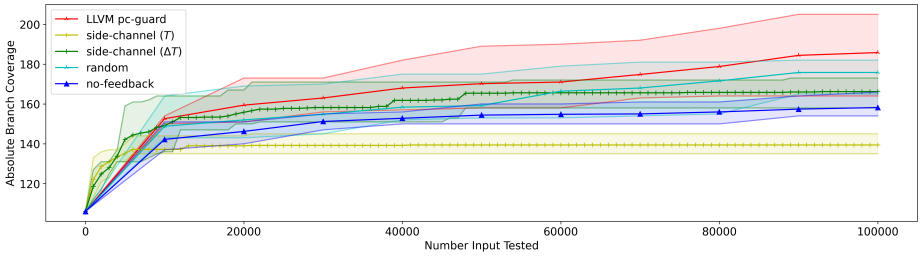
In Figure 7 and Table 3, we can see that the side-channel fuzzer provides improvements over a random fuzzer on the jpeg and zlib benchmarks. Not only the overall coverage is improved, but the coverage grows faster during the early moments of the campaigns which is interesting because it is a fast and strong indicator that the chosen metric is efficient. As observed on the emulation results (subsection 5.5), the greybox fuzzer outperforms the others. On zlib and libjpeg-v8 benchmarks (Figure 7a and Figure 7b), the T and ΔT side-channel fuzzer achieve significant coverage improvements. On average, the ΔT improves the coverage against a no-feedback fuzzer by 83% and 108% respectively. In the best iterations, the coverage improves by 87% and 127%. The T encoding improves the coverage by an average of 63% and 59%, respectively. On cJSON (Figure 7c), the SCA fuzzers perform a little worse than their emulated version. After 10,000 inputs, the T SCA fuzzer performs 11.88% worse than the no-feedback fuzzer and the ΔT fuzzer performs 5% worse than the random fuzzer.



(a) libjpeg-v8



(b) zlib



(c) cJSON

Fig. 7: Results of the SCA fuzzing engine.

6.6 Discussion and Future Works

As previously mentioned in subsection 5.5, the effectiveness of the feedback highly depends on the benchmark. Investigations on the cJSON benchmark show that the T side-channel fuzzer is stuck on the generation of integers and floats. Adding digits to an existing number increases the number of memory accesses in the parsing. Thus, any bigger number is categorized as interesting by the side-channel fuzzer. Despite the repetitive pattern detection capabilities of ΔT, it does not seem to be enough on this specific benchmark.

Fuzzing speed considerations have been deliberately left aside in this work. In our current setup, side-channel acquisition time limits the fuzzing throughput. In addition, for each input emitted by the side-channel aided fuzzer, the harness performs 5 executions on the target. The simplicity of our encoding schemes and side-channel analysis makes it a good candidate for real-time implementation.

We think an optimized side-channel acquisition and processing chain, such as a fully integrated FPGA solution, could drastically improve the fuzzing speed. The engineering efforts of such a design being high, it is therefore essential to first characterize the metrics and find the most relevant ones.

This work focused on improving branch coverage using side-channel information. However, the ultimate goal of blackbox fuzzing is bug detection. Analyzing coverage metrics is a first step in building an SCA fuzzer, as it is precisely measures path discovery. It would also be interesting to evaluate DuskFuzz against a bug-finding benchmark. Measuring SCA fuzzer's bug-finding capabilities is difficult. For instance, the well-known Magma [13] framework could bias side-channel observations through its instrumentations, while techniques such as Lava [7] are found to be unrealistic [17].

Finally, we tuned experimentally many parameters: window shapes, thresholds of side-channel metrics, number of acquisitions per input, etc. The parameters used in this paper are most likely suboptimal, but representative of a blackbox campaign. It would be interesting to conduct a comprehensive study on the effect of those parameters on the fuzzing quality. As well as the use of interrupt filtering [6] and AFL's calibration stage to reduce jitter and core's interrupt effect on more complex devices.

7 Conclusion

This work described DuskFuzz, a new framework for integrating side-channel information in an AFL-like fuzzer. The main idea is to build an event time-series from a side-channel source and integrate it in an AFL map. Two encoding schemes are proposed for integrating time-series into an AFL map. This approach abstracts away the side-channels source, enabling the use of various side-channel as input. Thus, the fuzzer can build feedback from physical side-channels like power consumption or electromagnetic-field, but also any information that can be transformed into a time-series (performance counters, micro-architectural events). The resulting fuzzer was validated on two case studies targeting a STM32F4 microcontroller. First, a fuzzing campaign on an emulated device which uses noiseless memory load and store traces as feedback. Second, a fuzzing campaign on a real side-channel analysis bench exploiting electromagnetic measurements. We demonstrated that the two encoding techniques proposed achieve improvements in branch coverage against a random fuzzer. On a real hardware target, improvements up to 127% of branch coverage were observed in the best case. Finally, the mitigated results on cJSON highlight that side-channel fuzzing can only work if the feedback contains enough information for differentiating corpus elements.

Acknowledgement. This work has benefited from a government grant managed by the National Research Agency under France 2030 with reference "ANR-22-PECY-0009".

8 Appendix

Algorithm 1 T encoding algorithm

Input: X: time-series, an ARRAY of N TUPLE of 2 (t_n, q_n)
Output: Y: map, an ARRAY of M values
1: $Y \leftarrow$ ARRAY[1:M] of 0
2: **for** $i \leftarrow 1$ to N **do**
3: $(t, q) \leftarrow X[i]$
4: $j \leftarrow t \bmod M$
5: $Y[j] \leftarrow Y[j] + q$
6: **end for**

Algorithm 2 ΔT encoding algorithm

Input: X: time-series, an ARRAY of N TUPLE of 2 (t_n, q_n)
Output: Y: map, an ARRAY of M values
1: $Y \leftarrow$ ARRAY[1:M] of 0
2: **for** $i \leftarrow 2$ to N **do**
3: $(t, q) \leftarrow X[i]$
4: $(t_{prev}, q_{prev}) \leftarrow X[i-1]$
5: $j \leftarrow (t - t_{prev}) \bmod M$
6: $Y[j] \leftarrow Y[j] + q$
7: **end for**

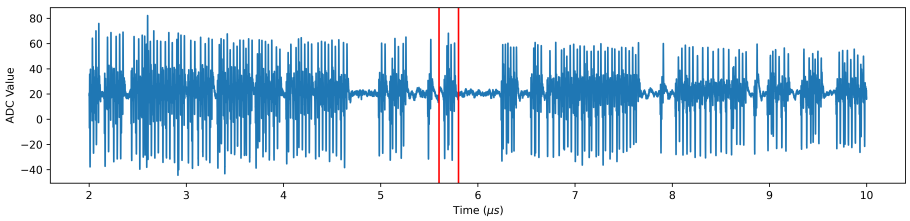


Fig. 8: Sample EM trace obtained during a JSON decoding (cJSON implementation).

References

1. Barredo, J., Petke, J., Clark, D., Blackwell, D., Eceiza, M., Flores, J.L., Iturbe, M.: GAFLERNA ahoy! integrating EM side-channel analysis into traditional

fuzzing workflows. In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering. pp. 550–554. ACM (2025). https://doi.org/10.1145/3696630.3728497
2. Basu, T., Aggarwal, K., Wang, C., Chattopadhyay, S.: An exploration of effective fuzzing for side-channel cache leakage. Software Testing, Verification and Reliability (2020). https://doi.org/10.1002/stvr.1718
3. Beckers, A., Balasch, J., Gierlichs, B., Verbauwhede, I.: Design and implementation of a waveform-matching based triggering system. In: Constructive Side-Channel Analysis and Secure Design (COSADE). Springer (2016). https://doi.org/10.1007/978-3-319-43283-0_11
4. Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W., Sun, M., Yang, R., Zhang, K.: Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In: Network and Distributed System Security Symposium (2018). https://doi.org/10.14722/ndss.2018.23166
5. Chen, Y., Bradbury, M., Suri, N.: Towards effective performance fuzzing. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 128–129. IEEE (2022). https://doi.org/10.1109/ISSREW55968.2022.00055
6. Dey, M., Yilmaz, B.B., Prvulovic, M., Zajić, A.: PRIMER: Profiling interrupts using electromagnetic side-channel for embedded devices. IEEE Transactions on Computers **71**(8), 1824–1838 (2022). https://doi.org/10.1109/TC.2021.3109457
7. Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R.: LAVA: Large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 110–121. IEEE (2016). https://doi.org/10.1109/SP.2016.15
8. Eisele, M., Ebert, D., Huth, C., Zeller, A.: Fuzzing embedded systems using debug interfaces. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 1031–1042. ISSTA 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3597926.3598115
9. Feng, X., Sun, R., Zhu, X., Xue, M., Wen, S., Liu, D., Nepal, S., Xiang, Y.: Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. p. 337–350. CCS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484543
10. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++ : Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association (Aug 2020). https://doi.org/10.5555/3488877.3488887
11. Goldack, M.: Side-channel based reverse engineering for microcontrollers. Master's thesis, Ruhr-Universität Bochum, Germany (2008)
12. Google: OSS-fuzz: Continuous fuzzing for open source software, https://github.com/google/oss-fuzz, original-date: 2013, access-date: 2023
13. Hazimeh, A., Herrera, A., Payer, M.: Magma: A ground-truth fuzzing benchmark. Proc. ACM Meas. Anal. Comput. Syst. **4**(3) (2020). https://doi.org/10.1145/3428334
14. Iyer, V.V., Thimmaiah, A., Orshansky, M., Gerstlauer, A., Yilmaz, A.E.: A hierarchical classification method for high-accuracy instruction disassembly with near-field em measurements. ACM Trans. Embed. Comput. Syst. **23**(1) (2024). https://doi.org/10.1145/3629167

15. jtpereyda: boofuzz: Network protocol fuzzing for humans, https://github.com/jtpereyda/boofuzz, original-date: 2015, access-date: 2025
16. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Annual International Cryptology Conference (1999). https://doi.org/10.5555/646764.703989
17. Lipp, S., Banescu, S., Pretschner, A.: An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 544–555. ISSTA 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3533767.3534380
18. Ma, X., Zeng, Q., Chi, H., Luo, L.: No more companion apps hacking but one dongle: Hub-based blackbox fuzzing of iot firmware. In: Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services. p. 205–218. MobiSys '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3581791.3596857
19. Maillard, J., Hiscock, T., Lecomte, M., Clavier, C.: Cache side channel attacks through electromagnetic emanations of DRAM accesses. In: 21st International Conference on Security and Cryptography (2024). https://doi.org/10.5220/0012813200003767
20. Malmain, R., Fioraldi, A., Francillon, A.: LibAFL QEMU: A Library for Fuzzing-oriented Emulation. In: Workshop on Binary Analysis Research (colocated with NDSS Symposium). BAR 24 (2024). https://doi.org/10.14722/bar.2024.23xxx
21. Marcussen, E.: Doona - network fuzzing tool, https://github.com/wireghoul/doona, original-date: 2022, access-date: 2025
22. Metzman, J., Szekeres, L., Simon, L., Sprabery, R., Arya, A.: FuzzBench: an open fuzzer benchmarking platform and service. ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2021). https://doi.org/10.1145/3468264.3473932
23. Nilizadeh, S., Noller, Y., Pasareanu, C.S.: DifFuzz: Differential fuzzing for side-channel analysis. In: IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 176–187 (2019). https://doi.org/10.1109/ICSE.2019.00034
24. Oh, J., Kim, S., Jeong, E., Moon, S.M.: Os-less dynamic binary instrumentation for embedded firmware. In: 2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII). pp. 1–3 (2015). https://doi.org/10.1109/CoolChips.2015.7158659
25. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: ACM SIGSAC Conference on Computer and Communications Security. pp. 2155–2168 (2017). https://doi.org/10.1145/3133956.3134073
26. Pham, V.T., Böhme, M., Roychoudhury, A.: Aflnet: A greybox fuzzer for network protocols. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 460–465 (2020). https://doi.org/10.1109/ICST46399.2020.00062
27. Redini, N., Continella, A., Das, D., De Pasquale, G., Spahn, N., Machiry, A., Bianchi, A., Kruegel, C., Vigna, G.: Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 484–500 (2021). https://doi.org/10.1109/SP40001.2021.00066
28. Riscure: Black box fuzzing with side channels. In: Hardweare.io (2022), https://hardwear.io/netherlands-2022/presentation/blackbox-fuzzing-with-side-channels.pdf

29. Schneider, T., Moradi, A.: Leakage assessment methodology. In: International Workshop on Cryptographic Hardware and Embedded Systems (2015). https://doi.org/10.1007/s13389-016-0120-y
30. Sesterhenn, E.: Using power side channel for fuzzing coverage, https://x41-dsec.de/news/2024/05/21/chipfuzz/, original-date: 2024, access-date: 2024
31. Sperl, P., Böttinger, K.: Side-Channel Aware Fuzzing, p. 259–278. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-29959-0_13
32. STmicroelectronics: RM0090 rev. 21 reference manual - STM 32F407/417 advanced ARM-based 32-bit MCUs
33. Su, K., Giraud, M., Borcherding, A., Krautter, J., Nenninger, P., Tahoori, M.: Fuzz wars: The voltage awakens – voltage-guided blackbox fuzzing on FPGAs. In: VLSI Test Symposium (VTS). IEEE (2024). https://doi.org/10.1109/VTS60656.2024.10538727