JMPscare: Introspection for Binary-Only Fuzzing

Dominik Maier, Lukas Seidel
TU Berlin
{dmaier@sect, seidel.1@campus}.tu-berlin.de
Both authors contributed equally to this paper

***Abstract*—Researchers spend hours, or even days, to understand a target well enough to harness it and get a feedback-guided fuzzer running. Once this is achieved, they rely on their fuzzer to find the right paths, maybe sampling the collected queue entries to see how well it performs. Their knowledge is of little help to the fuzzer, the fuzzer's behavior is largely a black box to the researcher. Enter JMPscare, providing deep insight into fuzzing queues. By highlighting unreached basic blocks across all queue items during fuzzing, JMPscare allows security researchers to understand the shortcomings of their fuzzer, and helps to overcome them. JMPscare can analyze thousands of queue entries efficiently, and highlight interesting roadblocks, so-called frontiers. Using this information, the human-in-the-loop improves the fuzzer, mutator, and harness. Even complex bugs, hard to reach for a generalized fuzzer, hidden deep in the control flow of the target, can be covered in this way. Apart from a purely analytical view, its convenient built-in binary patching facilitates forced execution for subsequent fuzz runs. We demonstrate the benefit of JMPscare on the ARM-based MediaTek Baseband. With JMPscare we gain an in-depth understanding of larger parts of the firmware and find new targets in this RTOS. JMPscare simplifies further mutator, fuzzer, and instrumentation development.**

I. Introduction

Emulators and dynamic binary instrumentation frameworks are very flexible tools. As some of these solutions introduce very little overhead, they can be used for performant feedback-guided fuzzing of binary-only code [7], [14], [16], [17], [22]. Modern binary-only instrumentation only lacks behind source-based instrumentation slightly, incorporating most modern mutation schemes and improvements. State-of-the-art binary instrumentation for fuzzing splits multi-byte comparisons at emulation time to be able to report successful hits of single bytes (*cmpcov* or LAF-intel [15]). It may even supply the fuzzer with feedback about compare results (*cmplog* [5], [9]) or inject address sanitization in an efficient way [7], [10]. Still, they are far from perfect and still get stuck on trivial checksums, complex floating-point calculations, and more complex formats like XML.

While fuzzers get smarter and smarter, being enriched with introspection, better instrumentation, and even symbolic execution [20], [27], it is still common for successful fuzzing campaigns to require lots of manual effort. On top of developing a functioning harness, the researcher collects common tokens and seeds, writes custom mutators [11] and grammars [2], [13] or may even actively guide the fuzzer with waypoints [4].

Especially for binary-only fuzzing, after the initial setup runs, getting deeper knowledge about the fuzzer's actual performance, and figure out shortcomings and roadblocks, can be a daunting task. Often, reverse engineers will look at the resulting coverage maps, debug hand-picked, interesting-looking inputs, and load up several collected traces in existing tools like Lighthouse [12] and Dragondance [1]. However, with thousands of queue entries for a fuzzer like AFL++, the tools reach their limits. Even if loading all traces as once succeeds, it is hard to spot interesting unreached jumps. A fuzzing queue, or corpus, is commonly used in feedback-based fuzzers, and collects interesting testcases that led to new coverage or otherwise new feedback during execution. We deem insights into this queue important to improve the overall harness and fuzzer performance. It shows us which parts of the harness the fuzzer explored and at which branches it got stuck. On complex targets, like a full-blown RTOS, a complete view of basic-blocks (not) reached would not provide the researcher with the same insights. No harness covers all basic blocks in the target: reaching some of the branches may rely on the target to be in a different state. Non-covered basic blocks may be reached with better harnessing and mutations or be completely unreachable. The human-in-the-loop has no chance to see this difference immediately just by looking at a coverage map.

To improve this situation, we developed JMPscare, an analysis toolkit that takes all traces in a fuzzer queue into account. The goal is to provide reverse engineers with full insight into their fuzzing campaigns. JMPscare efficiently uncovers never-taken branches, so-called frontiers. To guide the human-in-the-loop towards interesting frontiers, it performs a *Potential New Coverage* (PNC) analysis, surfacing frontiers that can lead to large new control flows if traversed., leading to additional corner cases. JMPscare offers a quick solution to overcome simple frontiers through a single click binary patching to force execution in this direction. With this, the fuzzer traverses any roadblocks, but triggers false-positives if the prior condition is direclty responsible for a crash. While JMPscare will work with any target, as long as a program counter trace of the executions can be acquired, JMPscare features a stand-alone trace collection with native *unicornafl* [11], *BaseSAFE* [17], and qiling [21] support. For the course of this paper, we evaluate JMPscare using the publicly available MediaTek harness and test case corpus of *BaseSAFE*. We are able to gather further knowledge about the harness that the original work could not collect and use, for the lack of JMPscare.

Workshop on Binary Analysis Research (BAR) 2021
21 February 2021
ISBN 1-891562-69-X
https://dx.doi.org/10.14722/bar.2021.23xxx
www.ndss-symposium.org

A. Contributions

The contributions of this work are as follows:

- We design, implement, and open-source JMPscare, an introspection toolkit suited for the analysis of fuzzer queues with thousands of traces at the same time, with support for *ARM32*, *MIPS*, and *X86_64* targets.
- JMPscare helps researchers to guide their fuzzers better through deep insights, useful analyses, binary ninja integration, and forced execution through binary patching.
- We evaluate JMPscare on the publicly available MediaTek Baseband harness, and released corpus of BaseSAFE [18].

B. Structure

In the following, we will discuss related work around fuzzing insights and forced execution. In Sect. III we will outline the design of JMPscare, including its included trace collector, the analyses, and its binary ninja plugin. Further, we evaluate it on the BaseSAFE MediaTek harness and corpus. In Sect. V, we discuss these results and further use cases for fuzzer development. After highlighting the next steps for JMPscare, we conclude with Sect. VII.

II. Background

The workflow we ultimately target with the binary-only JMPscare analysis is influenced by Ned Williamson's talk on *Modern Source Fuzzing* [26]. By the example of the XNU kernel source, he proposes, as an initial step, to patch out obstacles in source code for quick results, while at the same time working to improve harnessing for sound results. Although forcefully skipping over certain conditional jumps, read *checks*, can lead to subsequent false-positive bugs, the additional insights can help guide the researcher's manual analysis towards real bugs. To properly evaluate coverage data on binary-only runs, we may use Lighthouse [12] for IDA and Binary Ninja and Dragon Dance [1] for Ghidra. Both tools are not targeted towards fuzzing introspection or designed for the quick processing of thousands of files. In contrast to them, JMPscare's PNC analysis can guide the researcher towards very interesting frontiers. One of the few actual fuzzing introspection tools discussed publicly, although yet to be released, is the unnamed source-code fuzzer introspection tool by Aschermann and Schumilo [3].

In contrast to this almost unexplored field of fuzzer insights, *forced execution* has been well-researched. For forced execution, the target's control flow is artificially altered to overcome frontiers, by patching or instrumenting branches. Wilhelm and Chiueh load and emulate kernel rootkits for Windows while tracing multiple possible branches through forced sampled execution [25]. This allows them to explore the behavior of the rootkit with ease, even code paths, and functionality deeply hidden in the code. In the past, forced execution has also been applied to fuzzing, for example by Wang et al. [24]. Their fuzzer *TaintScope* uses dynamic taint analysis and symbolic execution and can bypass checksums by altering the control flow of the target. As early as 2007, Will Drewry and Tavis Ormandy proposed the use of control flow alteration for binary-only fuzzing, with their *Valgrind*-based *Flayer* [8]. While advanced for its time, Flayer does not work together with modern coverage-guided fuzzers. Its overall fuzzing performance is further hindered by the slow speed of Valgrind.

III. Design

In the following, we present the design and functionality of JMPscare. JMPscare is a collection of tools aiding researchers to unearth bugs hidden deeply within modern black-box binaries through fuzzing. Our goal with JMPscare was to create an easy-to-use analysis suite for multi-execution jump coverage introspection on a large dataset. It allows the reverse engineer to inspect fuzzing results, and specifically see, which important jumps were not taken by the fuzzer. As illustrated in Fig. 1, JMPscare contains tools for every step of the way: from trace collection to analysis and reasoning to visualization and binary patching.

A. Stand-Alone Trace Collection

While the JMPscare analysis is geared towards fuzzing, it does not operate on the fuzzer queue or related tools like *afl-showmap* of the original *AFL* [28] directly. Instead, JMPscare uses simple instruction-counter traces and is thus suited for the analysis of execution traces from various sources. The rationale behind this is simple: most binary-only fuzzers only hash edges. This hash may collide and hence cannot be reversed. A perfect 1:1 mapping of taken jumps will be impossible, making the default AFL coverage map useless. The instruction-counter traces used by JMPscare are simple to get for targets that support coverage-guided fuzzing. It is a list of all addresses executed for each input, or at least of all jump sources and targets, in order of execution. The JMPscare toolkit currently offers a Python library and a Rust crate to collect program counter (PC-)traces for *unicornafl* harnesses, *BaseSAFE* harnesses, and *qiling* harnesses. They all use the *Unicorn* multi-architecture CPU emulator framework [19] for binary-only instrumentation for black-box fuzzing with AFL++ [11]. Traces are generated for each input in the queue to be analyzed. To keep up with a large number of queue entries, JMPscare can make use of the harnesses forkserver to speed up trace collection.

As writing trace files constantly would slow down fuzzing, the trace collection should only be executed on-demand, independent of the actual fuzzing run. Whenever the researcher needs introspection, they will run the PC-trace collection on the current queue, then continue with the JMPscare analysis, discussed in the following.

B. Automated JMPscare Analysis

The automated JMPscare analysis is the core component of our contribution. It takes PC-traces and the binary as input. It is written in Rust and heavily relies on the widely-used *Capstone* [6] disassembler. At the moment, x86_64, MIPS, and 32 bit ARM (incl. thumb2) architectures are supported.

1) Finding Frontiers: By disassembling instructions at all traced addresses, the initial analysis pass determines which jumps were taken and which basic blocks were reached. The JMPscare analysis works quickly on thousands of traces, taking into account all basic blocks across multiple runs, taken from

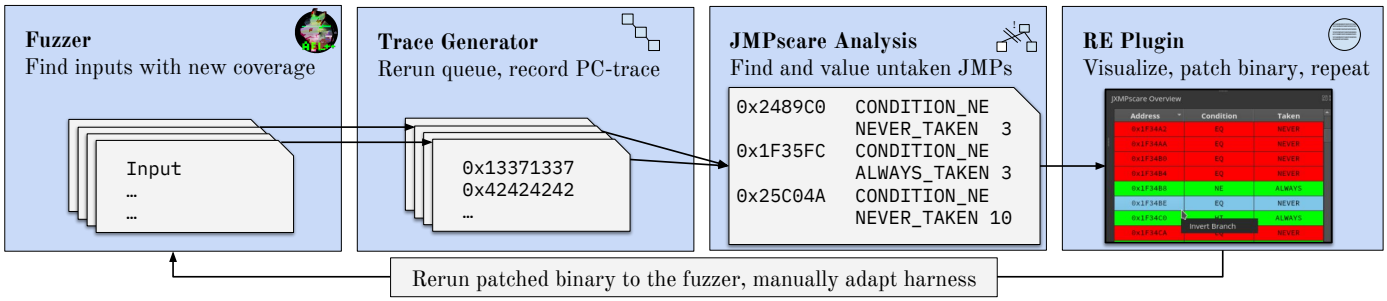


Fig. 1: The Trace Generator reruns the target for each interesting input the fuzzer found up to this point. The JMPscare Analysis then merges all found basic blocks, analyzes the taken and non taken branches, and weights blocks that were not reached. The researcher can display them in Binary Ninja, and choose to auto-patch jumps or adapt the harness for the next fuzz run.

the fuzzer queue. Our goal is to find basic block edges that the fuzzer was not able to overcome, so-called frontiers, see Fig. 2. To achieve this goal, all instructions for every collected PC-trace are first to read from the binary and disassembled using *Capstone*. For each conditional jump in a trace, we extract the jump's target address and insert the jump's information (e.g., condition, target address, instruction size) into a hashmap before the next instruction is disassembled. It is subsequently checked, whether the next address in the trace, i.e. the next executed instruction, matches the last jump's target address. If so, we register that the jump was taken (the condition was true), else the opposite. Upon finishing the parsing of all collected PC-traces, a search is performed to find conditional branches that were observered to have been executed for only one truth value. In the last step, noise reduction is performed by only retaining roadblock jumps in our list where following the non-observed branch does not immediately lead to a basic block that already has coverage. Example: We have encountered a `beq #0x2000` (*branch if equal*) and the condition was never met during all our fuzz runs, i.e. the jump was never taken. We only ever encountered the instruction's immediate successor, so if the branch was at address `0x1000` the next instruction in any trace would always be `0x1004`. On following the branch, we find that the basic block starting at address `0x2000` already has coverage because it was reached in at least one PC-trace on another control flow path. It, therefore, becomes uninteresting for further considerations.
Collecting all taken and non-taken jumps, we can reconstruct

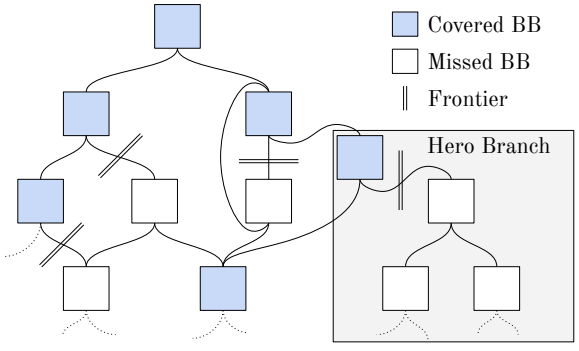


Fig. 2: Basic Blocks, as seen by JMPscare. White blocks have no coverage in the analyzed PC-traces.

all basic blocks the fuzzer reached over time, and uncover conditional jumps with unsatisfied conditions, so branches that were always, or never, taken across all PC-traces.

2) Potential New Coverage Analysis: Additionally, for supported platforms (ARM32 and *thumb* mode at the time), we perform the PNC analysis, a deeper analysis runs on the control flow graph, recreated by following branches, recursively disassembled with Capstone. In Fig. 2, the *hero branch* would get the highest PNC value, as none of the basic blocks behind the frontier were previously touched by any other input. The more new blocks we may reach, the more interesting it may be to the reverse engineer, hence we assign a high PNC that then gets displayed in the Binary Ninja plugin, see Sect. III-C. The PNC analysis assigns a value to each non-traversed frontier, guiding the analyst: a higher score is likely more important to be taken. Up to N edges for each of the previously found uni-directional jumps are traversed. The parameter is user-specified. Basic blocks are reconstructed again, visited blocks are tainted, and each previously unseen branch target gets registered and added to the list of edges to traverse next. After performing N traversals, we are left with information about potential new coverage hidden behind this roadblock jump, measured as the number of basic blocks without previous coverage. Each basic block can either have 0 (in case of a function return, cf. Listing 1), 1 (in case of an unconditional jump, cf. Listing 2) or 2 (in case of a conditional branch, cf. Listing 3) new edges.

Listing 1: Basic Block with no (resolvable) outgoing edge.

```
0x25ec74  add     r3, #0x174
0x25ec76  pop     {r4, r5, pc}
```

Listing 2: Basic Block with one (static) outgoing edge.

```
0x25d68a  mov     r0, r4
0x25d68c  ldr     r1, [pc, #0x1e0]
0x25d68e  b       #0x6c484c
```

Listing 3: Basic Block with two (conditional) outgoing edges.

```
0x1f34ae  cmp     r0, #0xa
0x1f34b0  beq     #0x1f3514
```

In the last iteration, i.e. after taking N jumps, unseen edges are still registered but not traversed. This leaves us with an upper bound of

$$b = 2^{N+1} - 1 \tag{1}$$

and a lower bound of 1 for the amount of possible new edges reachable in N jumps per initial roadblock jump. Function calls are usually treated as conditional jumps: the function, as well as the code after the call, will be treated as a new edge. For unresolvable branches, i.e. branches featuring register targets such as `blx r3`, the user has the option to guide the analyzer on how to treat them. Such jumps present no problem during the first analysis step, as we simply disassemble at a given address following an input PC-trace. However, resolving them during PNC analysis is not trivially possible as we would have to infer register contents during a static pass. The tool features a `weight` flag, specifying to what degree the observation of an unresolvable function call should be counted towards unseen basic blocks. With the default value of 1, each of these calls will be counted as one new block, even if the edge cannot be traversed. Specifying a weight of 0 leads to simply ignoring such calls. The user can also choose to make them count more. The reasoning behind this would be that in certain situations function calls may be of special interest, and huge weights allow the user to spot them more easily. While the function might have been seen before or might quickly terminate, it could also lead to a large previously undiscovered part of the binary. In the current design, the analysis tool expects and works on a single binary blob or code segment. This has the shortcoming that the tool would encounter a problem if it is presented with the PC-trace of e.g., an ELF, as it may contain jumps into code segments not being present in the loaded binary, e.g., into shared libraries.

3) JMPscare Analysis File: After JMPscare completes its analyses on all trace files in the queue, it outputs a single file. This file contains details about all uni-directional jumps. Every line describes one such jump, specifying the address in the binary, the jump's condition, whether it is taken always, or never. For PNC, it also includes the number of previously unseen basic blocks reachable within a user-specified amount of edge traversals, as discussed in Sect. III-B1, e.g.:

```
0x1172 CONDITION_LT NEVER_TAKEN 15
```

This generated analysis output summary can then be imported with the provided disassembler plugin for Binary Ninja [23], as discussed in the following.

C. Binary Ninja Plugin

The JMPscare toolkit comes with a Python plugin for use with the Binary Ninja reverse engineering software. While plugins for other popular RE suites, such as Ghidra, are planned, we chose Binary Ninja for our initial development. The software offers powerful and user-friendly scripting- and plugin environment. At the same time, the tool lays the focus on concise visualizations and was hence well-suited for what we wanted to achieve.

The plugin, see Fig. 3, provides a tabular overview of results previously obtained with the JMPscare analysis tool. Each row presents the user with all the information available for a certain address in the analysis summary and is colored green or red, depending on whether the jump was observed to have been always or never taken, respectively. Lines featuring found roadblock instructions are also colored respectively in the disassembly view, providing visual aid in navigating the graph view while looking into analysis results. The overview

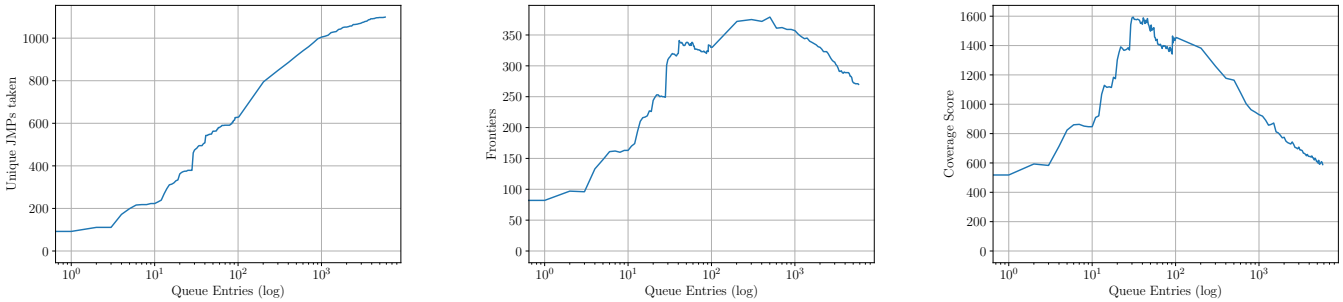
Fig. 3: JMPscare UI in Binary Ninja

can be used for quick navigation and automatic patching. Using Binary Ninja's built-in API capabilities, inverting a branch, e.g., from `beq #0x1000` to `bne #0x1000`, can be achieved via a single context-menu-click on any list entry, see the blue (activated) entry in Fig. 3. This alters the control flow quickly for forced execution. In the course of our work, we patched Binary Ninja to support this auto-patching feature for ARM's *thumb* mode, as it was formerly missing for this architecture. We subsequently opened a pull request to get this patch upstreamed.

IV. Evaluation

A. Analysis of Baseband Firmware PC-Traces

In the following, we show the application of JMPscare by analyzing 5902 traces obtained from executing the ARM firmware for MediaTek's Helio X10 (MT6795) baseband processor. The evaluated corpus was taken from the published BaseSAFE queue, fuzzing the baseband's `errc_event_handler_main` function, including various ASN.1 parsers. Fuzzing and emulation of the firmware were initially performed using *unicornafl* with AFL++. The following experiments were conducted with an Intel Core i7-1065G7 CPU @ 1.3 GHz with a maximum frequency of 3.8 GHz on a 512 GB M.2 NVMe SSD. To collect coverage information, the JMPscare-col Rust crate was plugged into the existing Rust harness [18]. Subsequently, the emulation was rerun on all 5902 fuzzing inputs, generating an equal amount of execution traces with one address per line corresponding to an executed instruction. The obtained traces feature 5860 executed instructions on average. As no instructions are added by the *unicornafl* instrumentation, this is the total count of target instructions. File input/output dominates execution time of JMPscare analysis runs. Averaged over 5 runs each, analyzing 5000 files took 17 seconds, and 2500 files took 8 seconds.



(a) Over time, the fuzzer takes less and less new jumps.

(b) As we limit the exploration depth, the JMPscare Frontier count increases when the fuzzer finds new paths. Once the fuzzer explored the target, the unexplored branches decrease

(c) As we only explore parts of the binary, the possible coverage initially increases as the fuzzer explores the binary further, but peaks after exploring most of the program.

Fig. 4: Observed Jumps and Coverage in relation to the amount of analyzed traces.

Analyzing 250 files never surpassed 1.2 seconds. In total, the 5902 execution traces contained 4161373 jumps, making up for about 12% of all encountered instructions. Of those, 1099 were unique. 270 were uni-directional, i.e. representing frontiers for the fuzzer that the harness was not able to overcome. Here, the branching condition was always true or always false in all the traces combined. PNC deep analysis was performed in three modes: with $N = 3$ jumps (default), 8 and 16 jumps. We observed 412, 590 and 1080 potentially new basic blocks reachable in N branches respectively. The top ten roadblock jumps with the most edges behind them offer between 14 and 340 new basic blocks without coverage in the analyzed traces, with the median being 120.

B. Fuzzer Exploration, Graphed

Additionally, we conducted an experiment to see how the numbers of uni-directional jumps and potential new edges being blocked by them correlate with the number of input PC-traces in the queue. For this, we picked one trace for each round, ran the JMPscare analysis on all available traces, and picked the next random trace, with a depth of $N = 8$ for PNC analysis. As we use a logarithmic scale, we continued to skip 100 traces after we had already collected a larger number of traces. Please note that execution traces were picked in an arbitrary order from the BaseSAFE queue, so results are not necessarily in the same in which the fuzzer discovered them originally. Figure 4 (a) shows the amount of unique frontier jumps in correlation to the number of execution traces on a logarithmic scale. As expected, after an initial increase, the growth stagnates as it becomes more and more difficult for the fuzzer to find inputs that lead to new program parts. In Figure 4 (b), the number of uni-directional jumps is illustrated. Additional execution traces reflect broader coverage, consequently containing more paths with more jumps. At some point, more condition-switching inputs than previously unknown jumps are found. The relation saturates before finally decreasing. In other words, in the beginning, a lot of totally new jumps are found and taken into one direction, until the right inputs are found to switch the condition and follow the second edge. From here on, the jump is no longer considered to be uni-directional, resulting in the observed decrease. Lastly, Fig. 4 (c) shows the Coverage Score, i.e. the number of previously unknown basic blocks behind all frontiers. As long as the analyzed traces themselves only provide marginal coverage, every new uni-directional jump may lead to huge increases in potential new coverage. Increased coverage through provided input cases leads consequently to earlier termination during traversal of unseen edges while performing PNC analysis, as the probability of encountering a block that already has coverage increases.

V. Discussion

In this section, we interpret our evaluation results. First, we discuss the direct positive impact JMPscare has on fuzzing harnesses, and specifically on the MediaTek Baseband harness. Then, we quickly describe an additional use-case: the improvement of fuzzers and instrumentation.

A. Improving Harnesses Through JMPscare

Naturally to the domain, many of the found frontiers turn out to be false positives. For example, the largest amount of new basic blocks was found to be behind a double roadblock. Following a full 4 byte compare (`cmp r3, r2`), we first encounter a branch if equal, which is never taken, followed by a branch if (unsigned) higher which is always taken. Upon closer inspection and reverse engineering of the code surrounding these branches, it turned out that this was the check for a so-called *Message-ID*, widely used throughout the firmware to guide incoming messages through different stages of baseband processing. As these Message-IDs were hardcoded within the fuzzing harness, it is clear why the code behind this check could not be reached. We analyzed recovered uni-directional jumps and prove that about 92% of them were either Message-ID checks or error handling, e.g., bounds checks, resulting in assertion failures. This shows that the BaseSAFE corpus is very well-explored, during weeks of fuzzing with compare coverage enabled. Very few reachable blocks remain behind complex frontiers. Given the nature of the target, a low-powered embedded device with a space-saving message format, this is unsurprising: the original experiment ran for multiple days, on multiple cores, and had *cmpcov*

activated [17]. As the evaluation shows, in Sect. IV-B, at an earlier point of time, a lot of blocks were not yet uncovered JMPscare, the original BaseSAFE paper could have taken a short-cut using JMPscare. By adjusting the aforementioned Message-ID in the harness, it is possible to reach different parts of the program. Although it is not surprising that fuzzing a different part of the program, which was previously purposefully ignored, leads to greatly increased coverage, the additional insights aid us in finding our next target. The `errc_event_handler_main` contains checks for more than 20 different Message-IDs of which not all may be equally interesting. The combination of JMPscare analysis and direct integration with a reverse engineering suite, such as Binary Ninja, enables us to effectively evaluate which target might be the most promising to fuzz next, leading to new coverage or large functions the quickest.

B. Basic Block Classification

Inspired by the insights and by further looking into the results of our analyses, we arrive at the following conclusion: After a fixed amount of fuzzing executions, given an arbitrary but fixed harness and initial state (e.g., a snapshot), every basic block in the target falls into one of the following categories:
1) *Reached:* The fuzzer has already produced an input to the harness which leads to or has a path containing the basic block. This block is in one of the traces.
2) *Reachable:* A basic block falls into this category if the fuzzer has the capabilities to reach this block in a reasonable time. It simply needs additional cycles to find input satisfying some conditions, or to find a certain, presumably long, path.
3) *Reachable Behind Frontier:* While still reachable in theory, a roadblock may not be solvable by the current fuzzer in any reasonable time. Manual aid, binary patching, or an improvement of the fuzzer itself, will be needed to overcome this frontier. This can be the case, when the control flow altering state becomes too complex, e.g., through deeply nested structs with multiple pointer indirections, or cryptography. JMPscare is especially valuable for reachable blocks behind frontiers.
4) *Reachable Altering Precondition:* Cases in which control flow to the block exists but are hidden behind state that cannot be changed by mutating the input. This may be the case if we start fuzzing from a snapshot, and the socket in question has not been opened before snapshotting. Another example are the Message-IDs we had hardcoded in the harness in Sect. V-A. To overcome this issue, we either have to change the harness, or start over with a different setup or snapshot. While altering the Message-ID is well in scope, it may be impractical to manually set up memory to mimic concepts such as whole file systems or open socket connections. For simple cases, forced execution can also help to reach these branches.
5) *Unreachable:* Of course, a block could be completely unreachable. In the case of a baseband, most parts of the baseband will never be reached from a given entry point. However, the block may become reachable with the first memory corruption the fuzzer finds.

C. Improving Fuzzers Through JMPscare

Improving the harness over and over throughout a fuzzing campaign, patching out checksums, adding tokens to the input, and so on, greatly improves the success chance. It is the norm to have a human in the loop, and it is still the major use case for fuzzers. However, we strive to make fuzzing available to a broader audience. This can only be achieved by keeping the complexity of the harness low. Instead, the fuzzer needs to be improved. For individual fuzzer and instrumentation engineering purposes, JMPscare can also deliver valuable insights. To give one example, *unicornafl* supports the splitting of compare operations for ARM and X86(_64), often called *cmpcov*, or *laf-intel* [11]. When fuzzing with *cmpcov* enabled, we see that the fuzzer traverses checks of two bytes more easily, which are quite common in baseband firmware, for tags and Message-IDs. On the flip side, the overall speed decreased slightly, from around $1.65k$ executions per second to $1.4k$ exec/s on the machine used during the evaluation. A deeper analysis of fuzzing performance goes beyond the scope of this paper. This anecdotal observation, however, suggests that researchers can directly measure and evaluate novel fuzzing mechanisms and their effect using the JMPscare analysis.

VI. Future Work

In the future, JMPscare can be extended in multiple directions. Using the JMPscare analysis, binary-only fuzzing can be improved, and recent advancements can be benchmarked in-depth, as briefly discussed in Sect. V-C. A straight-forward addition to our work will be support for additional reverse engineering platforms, such as IDA and Ghidra, and tracers to support additional instrumentation methods, for example, *qemu-mode*. During analysis, additional heuristics for interesting basic blocks can be implemented, and the GUI representation can even be enriched with taint and dataflow-data in order to provide the reverse engineer a better overview. Similarly, the JMPscare approach can be enriched with symbolic execution, in a similar fashion to *Driller* [27] and *QSYM* [27], however with additional human guidance. Apart from binary-only progress, the concept of JMPscare can be adapted to source-based fuzzing, for example, by analyzing *gcov* output for each test case, and then deducting basic block interestingness on a language level, or on the compiled binary.

VII. Conclusion

With the wave of published fuzzing research in recent years, it is surprising to see close to no existing tools for further introspection into the actual fuzzing runs. While fuzzers work on vast amounts of data and an increase of the overall performance can be statistically proven, deeper insight into the fuzz runs will immediately show the fine-grained roadblocks. With tools such as JMPscare, far better results can be achieved, as they aid researchers during manual reviews. While automated analyses improve steadily, putting the human in the loop can still greatly enhance the overall bug-finding performance. Forced execution allows the reverse engineer to potentially overcome hard challenges, such as checksums or cryptographic operations, without going through the trouble of rewriting the test case. At the same time, information gained through introspection via JMPscare can be used to aid a researcher in their decision where to continue fuzzing. In the case of fuzzing the MediaTek baseband firmware, we were able to make an informed decision on which message type, specified by the Message-ID, would be a promising next target without requiring large changes to our harness. We will use JMPscare to improve binary-only instrumentation in the future.

AVAILABILITY

JMPscare and all related source code is available open-source at https://github.com/fgsect/JMPscare.

ACKNOWLEDGMENTS

The authors would like to thank Jiska Classen for her valuable feedback and insights.

REFERENCES

[1] 0ffffffffh, "Dragon dance," Jan 2021, [Online; accessed 7. Jan. 2021]. [Online]. Available: https://github.com/0ffffffffh/dragondance

[2] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "NAUTILUS: fishing for deep bugs with grammars," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[3] C. Aschermann and S. Schumilo, "What the fuzz," Black Hat Europe 2019, Dec. 2019. [Online]. Available: https://www.blackhat.com/eu-19/briefings/schedule/#what-the-fuzz-18031

[4] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1597–1612.

[5] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[6] Capstone, "The Ultimate Disassembly Framework," Nov 2020, [Online; accessed 9. Jan. 2021]. [Online]. Available: https://www.capstone-engine.org

[7] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1497–1511.

[8] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," in *First USENIX Workshop on Offensive Technologies, WOOT '07, Boston, MA, USA, August 6, 2007*, D. Boneh, T. Garfinkel, and D. Song, Eds. USENIX Association, 2007. [Online]. Available: https://www.usenix.org/conference/woot-07/flayer-exposing-application-internals

[9] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 1–13.

[10] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with qasan," in *IEEE Secure Development, SecDev 2020, Atlanta, GA, USA, September 28-30, 2020*. IEEE, 2020, pp. 23–30.

[11] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[12] gaasedelen, "Lighthouse - a code coverage explorer for reverse engineers," Jan 2021, [Online; accessed 7. Jan. 2021]. [Online]. Available: https://github.com/gaasedelen/lighthouse

[13] R. Gopinath and A. Zeller, "Building fast fuzzers," *CoRR*, vol. abs/1911.07707, 2019. [Online]. Available: http://arxiv.org/abs/1911.07707

[14] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/gustafson

[15] laf intel, "Circumventing Fuzzing Roadblocks with Compiler Transformations," Aug. 2016, [Online; accessed 11. Jan. 2021]. [Online]. Available: https://lafintel.wordpress.com

[16] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *13th USENIX Workshop on Offensive Technologies, WOOT 2019, Santa Clara, CA, USA, August 12-13, 2019*, A. Gantman and C. Maurice, Eds. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/maier

[17] D. Maier, L. Seidel, and S. Park, "BaseSAFE: baseband sanitized fuzzing through emulation," in *WiSec '20: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Linz, Austria, July 8-10, 2020*, R. Mayrhofer and M. Roland, Eds. ACM, 2020, pp. 122–132. [Online]. Available: https://doi.org/10.1145/3395351.3399360

[18] ——. (2021, Jan) BaseSAFE: Open source framework. [Online; accessed 7. Jan. 2021]. [Online]. Available: https://github.com/fgsect/BaseSAFE

[19] A. Q. Ngyuen and H. V. Dang. (2020) Unicorn: Next generation CPU emulator framework. [Online]. Available: http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf

[20] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[21] qilingframework, "Qiling advanced binary emulation framework," Jan 2021, [Online; accessed 12. Jan. 2021]. [Online]. Available: https://github.com/qilingframework/qiling

[22] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 19–36. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ruge

[23] Vector 35 Inc., "Binary Ninja," Jan 2021, [Online; accessed 7. Jan. 2021]. [Online]. Available: https://binary.ninja/

[24] T. Wang, T. Wei, G. Gu, and W. Zou, "Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 2, Sep. 2011. [Online]. Available: https://doi.org/10.1145/2019599.2019600

[25] J. Wilhelm and T.-c. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *Recent Advances in Intrusion Detection*, C. Kruegel, R. Lippmann, and A. Clark, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 219–235.

[26] N. Williamson, "Modern source fuzzing [talk]," OffensiveCon19, Feb. 2019.

[27] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[28] M. Zalewski, "American Fuzzy Lop - Whitepaper," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016.