Fuzzing and Symbolic Execution for Multipath Malware Tracing: Bridging Theory and Practice via Survey and Experiments

MARCUS BOTACIN, Texas A&M University, USA

In real life, distinct runs of the same artifact lead to the exploration of different paths, due to either system's natural randomness or malicious constructions. These variations might completely change execution outcomes (extreme case). Thus, to analyze malware beyond theoretical models, we must consider the execution of multiple paths. The academic literature presents many approaches for multipath analysis (e.g., fuzzing, symbolic, and concolic executions), but it still fails to answer *What's the current state of multipath malware tracing?* This work aims to answer this question and also to point out *What developments are still required to make them practical?* Thus, we present a literature survey and perform experiments to bridge theory and practice. Our results show that (i) natural variation is frequent; (ii) fuzzing helps to discover more paths; (iii) fuzzing can be guided to increase coverage; (iv) forced execution maximizes path discovery rates; (v) pure symbolic execution is impractical, and (vi) concolic execution is promising but still requires further developments.

Additional Key Words and Phrases: This is the public version of the paper.

ACM Reference Format:
Marcus Botacin. TBD. Fuzzing and Symbolic Execution for Multipath Malware Tracing: Bridging Theory and Practice via Survey and Experiments. *Digit. Threat. Res. Pract.* 0, 0, Article 0 (August TBD), 33 pages. https://doi.org/TBD

1 INTRODUCTION

During a malware run in a tracing solution, the taken paths might vary due to internal (e.g., logic bombs [88]) and external factors (e.g., natural randomness [82]). Thus, to provide a comprehensive view of malware, an analyst should explore distinct execution paths. This might include purposely forcing the software to execute via paths that were not initially naturally observed [64]. The academic literature is rich in solutions to lead executions via interesting paths (e.g., fuzzing [36], forced [64], symbolic [93], and concolic [80] executions), but they are mostly very focused on vulnerability discovery, with limited attention given to malware analysis, a field which could largely benefit from these techniques. However, to make these techniques viable to be applied in the malware analysis domain, more developments might be required.

Unfortunately, we have currently no Systematization of Knowledge (SoK) about the solutions for multipath malware execution, such that literature is still unable to draw a clear landscape of current tools capabilities and analysis practices and thus the developments required to move the field forward. In this work, we bridge this gap by **(1)** surveying the current literature to point out the already identified good practices and the existing developments gaps; and **(2)** putting these findings to the test by implementing the literature approaches and testing them against a dataset of 21K real malware samples [12].

We reimplemented one solution representative of each multipath tracing technique and deployed them in parallel sandbox instances to run the same samples in the multiple techniques to compare their result. The 21K unique malware samples resulted in more than 21M traces, which allows us to pinpoint where each technique succeeds and struggles. Our goal is not to compare which solution is the best–we consider that all of them are not mature and require new developments to be practical–but to pinpoint what are the developments that each solution requires and how they can benefit the practice of malware analysis in concrete cases.

Author's address: Marcus Botacin, botacin@tamu.edu, Texas A&M University, College Station, Texas, USA.

2576-5337/TBD/08-ART0 $15.00
https://doi.org/TBD

Our results show that (i) natural variation is frequent during tracing even when no fuzzing technique is used; (ii) fuzzing is effective in discovering more paths even in samples that initially appear to have a very stable main execution path; (iii) fuzzing can be guided to increase coverage, but it limits the diversity of the discovered paths; (iv) forced execution achieves the greatest path discovery rates by exploring all possible paths, even though it costs a lot in terms of performance; (v) pure symbolic execution is impractical, both in terms of performance as well as in terms of threat models, as it currently fails to handle packed samples; and (vi) concolic execution is promising approaches, but still requires further developments, such as support for a larger extent of the Windows APIs. After that, we consolidate the results from the multiple runs to show what causes malware to present diverging execution paths. This is the largest scale evaluation of diverging malware behavior presented in the literature–scaling the previous hundred-sized reports to a thousand-sized dataset.

In summary, our contributions are twofold:

- A systematization of knowledge about multipath malware execution via a literature review;
- A systematization of practices via experiments comparing the multiple literature techniques.

This work is organized as follows: In Section 2, we motivate the need for multipath execution; In Section 3, we present a literature review on existing techniques; In Section 4, we present an empirical evaluation of the surveyed techniques; In Section 5, we show how different multipath techniques impact detection applications; In Section 6, we discuss our findings and this work's limitations; Finally, In Section 7, we draw our conclusions.

2 BACKGROUND & MOTIVATION

In this section, we (i) motivate the need for multipath execution for proper malware characterization and for (ii) further developments towards multipath malware tracing frameworks, and (iii) exemplify how different execution paths manifest into analysis trace results.

2.1 The need for multipath tracing

Why does one need multiple executions? Many analyses ideally assume that a single execution is enough to characterize an application (wrt., performance [86], security, and so on). However, real-world application executions are not very deterministic. Thus, a proper application characterization is only possible by observing multiple runs of it. Academia is starting to bridge this gap and to make systems more practical [14]. The uses of multiple execution approaches are varied, especially for security. For instance, one might: (i) force paths to *"buy time"* for complex Machine Learning (ML) scans [82]; (ii) test paths to find invariants for signature generation [13]; and (iii) explore multiple paths implementing network activities to characterize malware [91]. We here focus on multipath malware tracing.

Why does one need multipath execution for malware handling? Analyzing malware samples is different from analyzing other software. Unlike goodware, malware is allowed to *"break the laws"*–not follow standards–, which complicates analysis tasks, as malware samples do not provide any guarantees about invariant execution. Thus, different executions among multiple paths are a significant side-effect of malware coding. We found that the key roots for deviating runs are:

(1) **Natural randomness and Buggy constructions.** Whereas sophisticated malware is well-coded, most samples are created in a *"best-effort"* manner, often skipping error checking and corner cases handling [21] (e.g., resource unavailability, mutexes [82], and so on). Thus, natural failures will result in distinct explored paths and outcomes.
(2) **Evolving payloads.** malware downloaders [70] often mutate their payloads (e.g., via server-side polymorphism [29]) to distribute a distinct payload at every request. Thus, multiple runs will result in distinct outcomes.

(3) **Takeover resilience.** Malware might also change the path to reach a payload or C&C. To avoid the takeover of their malicious domains, some malware (e.g., botnets [81]) rotate the contacted domains and IPs across executions to avoid revealing all its secrets at once. Thus, an analyst must run the same sample multiple times to uncover the multiple domains.
(4) **Logic bombs.** Not all malware samples will immediately act maliciously. Some samples are armored to act maliciously only under certain circumstances [34, 84, 88] (e.g., on a given day, after some time, and so on) that are called logic bombs [90]. They can only be defused by exploring multiple paths to identify the truly malicious execution.
(5) **Environment-sensitiveness.** Malware might also be armored with specific triggers that depend on the context (e.g., OS versions, usernames, world regions, keyboard languages). These called environment-sensitive malware [79] fingerprint the environment before their run. Again, the analysts must explore multiple paths to find a truly malicious one.

2.2 The evolution of multipath tracing

From goodware to malware. Though malware analysis requires multipath execution and the literature has solutions for that in other contexts, the number of reports of multipath malware executions is still limited. The uses of multipath execution strategies are mostly often focused on the analysis of goodware samples and in bug-finding tasks. Whereas analyzing these cases is essential, we believe that overlooking the malware scenario is a problem because the developments in those fields might not generalize for the malware case.

Many literature works analyzed the challenges and limitations of existing approaches for multipath execution for offensive purposes [77]. However, most works do not dive into details about the use of these solutions for defensive aspects. This limits malware analysis generalization, as some research works investigate the challenges of concolic execution and the prevalence of its limitations only for goodware samples [67]. It is key to measure these same limitations for malware samples because they are prone to purposely explore the limitations of analysis systems. It is plausible that the limitations for malware analysis were greater than for goodware samples, which tend to be more *"well-behaved"*.

Some previous works compared the performance of the multiple approaches (e.g., multiple symbolic executor's Intermediate Representations (IR) [65]), but they did not discuss if the IRs support Windows functions, which is key for malware analysis. Also, whereas tools were compared for the bug-finding task [28], the same comparison has not been presented for malware analysis, a gap we bridge in this paper.

One can find only a few research works on multipath malware execution. Whereas their results are promising, they still present multiple limitations. For instance, a non-academic study showed results of fuzzing malware network communication for bug finding [78]. Whereas the results are illustrative of hidden paths in malware samples, this study is not systematic, a gap to be bridged. Similarly, a study [4] published while we conducted this survey showed how Windows malware samples are prone to present execution variations (one needs at least 4 distinct runs to characterize a sample). However, this study was completely based on malware stimulation and did not evaluate the effect of distinct techniques on it, another gap to be bridged.

The First approaches for multipath malware execution. They initially consisted of running the same sample multiple times, via either record-and-replay [7], that compare each execution with a reference, or parallel virtual machines [92], each one running a distinct environment to highlight context dependencies. The tools allowed the analyst to revert a snapshot at a decision point to inspect the divergence cause [60] or to align the diverging trace with the reference one [44]. Their major drawback is that the stimulation of the malware samples is uniquely provided by the distinct environment itself, with no code coverage guarantees. These solutions were often used to evaluate the effect of running code on emulators and/or virtual machines. In sum, the execution in the tested

environment was compared with the execution in the bare-metal system taken as a reference. Currently, there are solutions for this purpose available both for desktop [44] and mobile enviroments [2].

Over time, the stimulation process became more sophisticated, by either adding manually-defined behaviors to be checked [44] or allowing the configuration of distinct environments. For instance, parallel virtual machines might run at distinct speeds to detect timing bombs [30]. Despite these advances, the simulation capabilities are still reduced in comparison to modern approaches like symbolic execution.

A key contribution from the first works was to highlight the natural variations involved in comparing malware traces and summarize the trace comparison requirements. Previous work pointed that distinct executions of the same malware involve a distinct number of executed instructions due to natural clock tick variations and process scheduling. Thus, trace lengths must be normalized [52]. Also, resources might have distinct names in distinct environments (e.g., usernames, default locations, environment variables) and they should not be considered in deviations. Thus, paths and timestamps should be normalized [2]. Previous work showed that network traffic must be replayed to avoid deviation due to DNS timeouts [48].

Some works focused on trace alignment since properly aligning a trace is key to understanding if (and where) two executions diverge. The alignment problem is performance-intensive: Previous work demonstrated that aligning two long traces might take hours [42]. Many strategies were proposed to mitigate alignment issues (e.g., genetic algorithms [47]) but efficient alignment remains an open problem.

2.3 Multiple execution paths in analysis trace results

To understand how good a technique is in finding execution paths, it is key to understand how the execution paths manifest in trace results. We here present a didactic example to clarify the concepts used in this paper. To that, consider the didactic malware sample shown in Figure 1, which presents different behaviors depending on the execution environment. On it, internal malware constructions are displayed in gray and externally-visibile API functions are displayed in coral.

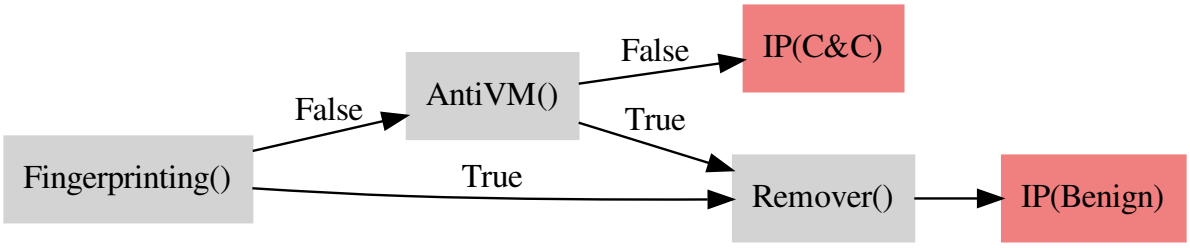

Fig. 1. **Flow graph of a malware.** Different execution paths depending on internal code constructions (in gray) lead to different API functions (in coral) appearing in the trace and different block and edge coverage values to be reported.

If one only analyze this sample in a sandbox without additional stimulation, the sample might fail due to the initial sandbox fingerprinting and then connect to the benign IP address as a deceptive behavior. If external stimulation is provided, the fingerprint and anti-VM checks could fail and the sample reveals its malicious intent: the communication with the Command-And-Control (C&C) servers. The benign and C&C IPs are the artifacts observed in API traces. Whenever this paper mentions different API traces, we refer to the difference between obtaining the benign or the C&C IPs.

Having different traces at the API level allows spotting splitting behavior but it is not enough to explain the divergence root causes. To that, it is key to look at the internal binary constructions. The internal binary paths can be measured via different metrics, such as via the blocks it traverses. In the previous evasion case, the execution

path was via the `Fingerprint` and `Remover` functions. In the previous success case, it was via the `Fingerprint` and `Anti-VM` functions. It shows that different blocks were triggered in each case. In some cases, however, the same blocks might be reached, but via different ways, which also ensures path diversity. The sample could also evade analyses via the `Fingerprint`, `Anti-VM`, and `Remover` path. While the `Remover` functionality is present in both paths, it was reached via different edges. Thus, when measuring the different execution paths, we refer to the proportion of reached blocks and edges as block and edge coverage. The greater the coverage, the more we understand the binary. In this work, we adopt all the above metrics to evaluate the investigated solutions.

3 LITERATURE REVIEW

In this section, we present a review of the current state-of-the-art techniques for multipath exploration, highlighting their pros and cons. For our review, we relied on a previous literature review on the malware research published in the last 20 years [14] and filtered from it the papers that mentioned multi-path execution. This literature review is the most comprehensive literature on malware analysis paper up-to-date, and therefore it was considered a good starting point. To enrich its results, we applied the snowball methodology to query via Google Scholar the most recent papers that cited the papers originally included in this literature review, also having as inclusion criteria the ones that mentioned multipath malware execution. The papers are below presented broken down by technique type.

3.1 Fuzzing

Fuzzing is the process of adding randomness to an application run to maximize the variations a program exhibits. This is key for many security tasks, such as bug finding, that benefit from the caused variations to reach the problematic states with a greater probability than simple luck, as in ordinary runs. Malware analysis systems could also benefit from fuzzing-induced randomness to maximize the variations within the analysis environment. A fuzzer would be more efficient in adding randomness to the system than adding multiple parallel virtual machines, as done by the first works in the field.

However, despite these possibilities, the initiatives to fuzz malware using standard fuzzers are still limited. Whereas we can find in the literature multiple fuzzing solutions (e.g., `AFL` [37] and its variations [33], `VUzzer` [69], `Angora` [26], and so on), we could not find a single work in the literature that purely applies conventional fuzzing strategies to the malware analysis problem. We believe that the application of fuzzing would be good for forcing samples to exhibit the natural variation cases, to find unhandled cases (e.g., function invocations and returns without checks), and test the sample's resilience (e.g., by checking if the failure of a function call with a given IP as an argument results in the sample calling the same function with a distinct IP as an argument in the future).
Fuzzing for bug discovery vs. malware analysis. Whereas it seems that the malware analysis field is missing the opportunity to benefit from unmodified fuzzers, researchers in the field also have many opportunities to customize existing fuzzers: Due to the nature of the malware analysis problem, fuzzers must be adjusted to fuzz malware in addition to helping with the bug-finding problem. A key difference between the two problems is the nature of the fuzzed objects: Whereas fuzzers for offensive security often operate over the application input (e.g., files, streams, and so on), malware samples are often self-contained applications, having no input files. Malware samples also rarely present arguments to be fuzzed. Even when they do so, the sample's arguments are usually filled by malware loaders, typically unavailable at fuzzing time. Instead, the greatest information source for malware execution is the execution environment itself, which is made available to the malware samples via the system's and library's APIs.

Typical API fuzzers work by changing the API call arguments across the multiple runs [11]. However, this approach is not suitable for malware analysis because function arguments are key for malware operation (e.g., they might be files, IPs, and so on) and even constitute important Indicators of Compromise (IoCs) for malware

characterization. Thus, malware fuzzers should not change the sample's arguments, but change what they do with the arguments. In this sense, a key change to fuzz malware is not to fuzz function arguments but to fuzz the function returns. The rationale is that by changing the information the environment provides via the distinct return codes, the malware samples will present distinct behaviors. Whereas this rationale was already applied to the first works in the field using multiple VMs, fuzzing solutions would have the significant advantage of performing the variations in place.

Implementation Limitations: Fuzzing Tools. We identified one practical aspect that might be responsible for the scarce number of malware fuzzers in the literature: The targeted Operating System (OS). Whereas most malware samples target Windows, due to its popularity, most fuzzers are developed for Linux. This difference imposes limitations not only on the nature of requiring one to port a solution from Linux to Windows but also on the nature of completely changing the fuzzer architecture, as some concepts are different on the two OSes. For instance, a typical fuzzer for Linux (e.g., AFL) implementation decision is to leverage the *copy-on-write* capabilities of the Linux kernel to split states via the *fork* syscall. Therefore, a new process is launched whenever a program state must be branched without imposing the need for re-running the process until reaching the same point, but also without imposing the overhead of copying the whole context every time. This same implementation strategy is not possible on Windows, which does not have an analogous to the *fork* syscall. Fuzzers for Windows (e.g., WinAFL [66]) must use the *CreateProcess* function instead. This function creates a process from an empty state, thus fuzzers cannot benefit from the *copy-on-write* mechanism, which slows down fuzzing.

Fuzzing for malware also presents performance bottlenecks due to its almost serial execution characteristic. Whereas most goodware samples can be fuzzed in parallel, as their executions tend to be independent, most malware runs must be serial, as a previous execution tends to influence the next one (e.g., many samples check for previous infections). This happens because whereas goodware applications rarely leave system-wide artifacts that significantly affect future executions (e.g., file deletions, global mutexes, and so on), malware samples often cause significant system changes, such that concurrent executions of the same malware tend to be limited by those conflicts. Thus, whereas a typical goodware fuzzer forks a process many times during a run, a malware fuzzer often reverts the system to a clean state (e.g., VM snapshot) to run a new analysis. This shows that an ideal malware fuzzer should be integrated with a sandbox and isolation mechanism. Developing this type of solution is still an open problem.

Evaluation Limitations: Coverage. A challenge for future uses of malware fuzzers (and current uses of other multipath approaches) is to evaluate the executed paths and even to know when the samples were fuzzed enough. Many researchers investigated the problem and created metrics for that, of which coverage plays a central role [31]. Coverage can be defined in a high-level way as the ratio of the multiple paths covered during an analysis. The ratio can be more fine-grained measured for instance as the number of visited basic blocks and/or edges of a Control Flow Graph (CFG).

As the number of states that a program might present is very large (with exponential growth at every branch), some fuzzers even guide their execution at every branch decision to maximize the coverage (coverage-guided fuzzing). The implicit assumption of this strategy is that covering large portions of binary results in a more representative view of the application. Whereas this might be true for some cases, there are many cases in which this assumption fails: Previous work showed that the fuzzer that achieves the greatest coverage might not be the one that discovers more bugs [20].

The same finding might not be true for malware. Malware binaries might have large portions of code that are useless (dead code). These code regions are added to the binaries by the attackers to confuse detection mechanisms and analysts. These code regions might also be derived from trojanized goodware applications that became malicious due to a patch/recompilation. In this case, large parts of the binary code become unreachable, but they are kept in the file to make it resemble a goodware application. The problem of evaluating malware binaries in the presence of dead code has already been pointed out in other security contexts (e.g., decompilation [15]). In

this task, the fraction of recovered code depends on the amount of deadcode in the binary. Unfortunately, we did not find literature works addressing the coverage problem for malware fuzzing, which remains an open problem.
Future threats to malware fuzzing. As fuzzing for malware progresses, attackers will likely implement attempts to defeat the technology. Whereas such attempts are not currently commonly found in the wild, a literature review can predict some emerging technologies as candidates for future malware instrumentation. More specifically, the recently published works on anti-fuzzing techniques [38, 43] constitute potential interest for malware authors, as they can make the fuzzing of binaries more complicated and time-consuming.

3.2 Forced Execution

Forced execution is a variation of fuzzing. Whereas in a typical fuzzer new variation points are introduced to add randomness to the system, the triggering of these variation points is probabilistic. In the forced execution approach, the execution state is always split at every instrumentation point, thus increasing coverage. Whereas pure fuzzing approaches are rare in the literature, one can find more examples of forced execution. This approach seems more popular in the Android platform [1, 72, 87], even though x86-based solutions can also be found. The most noticeable example of forced execution in the desktop environment is likely XForce [64], a solution implemented on Intel PIN to invert the branch conditions of malware binaries and thus increase coverage.
Implementation Challenges. A typical implementation of the forced execution approach consists of changing the values of API function returns [87], as discussed in the fuzzing section. However, the definition of the return values might not be straightforward, depending on the code construction to be addressed. Whereas `IF-ELSE` constructions are easily addressed by simply inverting (negating) the function return value, `switch` constructions are challenging. Researchers have to develop alternatives to handle these cases, including transforming the `switch` statements into `IF-ELSE` constructions [87] to allow the application of the default heuristic.
Computation Limitations: Which paths to force? A key decision of a forced execution approach is which paths will be forcedly executed (the variation points). Whereas the idea of forced execution is to explore more paths than fuzzing by deterministically forcing more paths, splitting states at every branch is exponential, and thus impractical. It leads to the path explosion problem, which is a major limitation for fuzzers and other multipath approaches to reaching deep binary states [80]. Whereas the path explosion phenomenon was already a problem for fuzzers, it is exacerbated in forced execution.

Thus, explored states must be pruned to a reasonable number. Due to this requirement, many forced execution approaches are focused on reducing the exploration to a minimum required set to still reach high coverage [1]. A common reduction strategy is to prune the exploration of libraries, which are often complex and present multiple internal states, such that forcing internal states might lead to many complications for forced execution [72]. To mitigate the path explosion problem inside the libraries, a possible strategy is to force only the paths resulting from the exploration of the main malware code and not in the invoked external library functions.

Whereas we can find many works in the literature that evaluate pruning strategies for forced execution and fuzzing for offensive purposes (e.g., bug finding), we could not find examples of systematic evaluations of pruning strategies for the malware exploration problem. Existing works on forced malware execution seem to adopt the implicit assumption that the best strategies for offensive purposes are also the best strategies for malware exploration. However, the validation of this hypothesis is still an open problem.

3.3 Symbolic Execution

Whereas fuzzing and forced execution are reasonably effective strategies for multipath execution, they present a major drawback: there is no guarantee that all (or most) paths will be explored. For instance, even though one changes the return of an API function call, if the new value is not within the range expected by the code, its above paths will not be explored. Similarly, even if some branches are modified (e.g., conditions inverted),

some deep program states can only be reachable via specific combinations of branch inversions. Moreover, these approaches require the concrete execution of the program, which might be resource-consuming.

Alternatively, symbolic execution does not operate by concretely executing the binary code, but by executing a symbolic version of it, i.e., symbolic executors operate by lifting the binary code to an intermediate representation that mathematically models the relations between the variables. This allows the symbolic executor to identify which constraints lead to the exploration of distinct paths. Due to the symbolic nature, paths can be explored in parallel and states can even be merged if particular branches end up not affecting the path exploration. Due to these capabilities, efficient symbolic execution of malware would be interesting for analysis purposes, as it would allow one to discover precisely the constraints for a malware sample triggering its malicious behaviors.

As for the previous cases, the literature on symbolic execution is richer in the bug-finding task than in malware analysis. In this case, however, the malware analysis task was addressed by at least a few works dedicated to malware analysis [50]. The strategy adopted in most of these works is to convert the function returns into symbolic representations to gather knowledge about the expectations malware samples have about the environment [3]. In some cases, the symbolic analysis is directed towards specific malware execution steps, such as forcing malware execution via paths derived from successful network interactions [5]. Despite these motivating examples, symbolic execution for malware analysis has still a long path to go before becoming practical, as below detailed.

A multitude of symbolic execution tools. As for traditional fuzzing, one can find a multitude of solutions described in the literature for symbolic execution, either in its pure form or in its variations, such as concolic execution (e.g., `angr` [77], `Triton` [74], `Klee` [22], `BAP` [19], `SAGE` [36], `Metasm` [41], `Miasm` [57], `Triton` [68]. `Mayhem` [25], and `S2E` [27]). Another common finding with the fuzzing case is that whereas many of these solutions are tuned for specific purposes (e.g., bug finding), no solution is tuned for malware analysis.

Symbolic execution for malware analysis: where to start? As for the previous approaches, the use of symbolic execution for malware analysis presents some particularities and challenges. Some assumptions made in other contexts do not hold for the malware analysis task. A key critical assumption that must be revisited is where to start the exploration process. In most solutions, the symbolic executor is positioned at the function entry point, which is typically the main function, identified via symbols. Whereas some malware analysis approaches also adopt this strategy [50], this might present limitations for the analysis of stripped malware binaries, which do not present symbols. We could not find discussions in the literature about how to overcome this limitation.

Symbolic execution for malware analysis: where to go? And where to stop? Another key assumption to be revisited is where to stop path exploration. Analysts often do not have infinite time to explore all possibilities, so some stop criteria must be established. The typical symbolic executors targeting bug-finding have a clear stop point criteria: crash sites. This makes sense as a crash site indicates that an application has a bug or unhandled case and finding it was the fuzzer goal. Malware analysis does not have such clear criteria, as a successful malware execution tends to not crash but to act maliciously. Thus, researchers must develop their own assessment criteria.

The stop criteria problem is related to the `path priorization` problem. During an exploration, not all paths are of the same interest to the analyst. Certainly, whereas some paths lead to the target criteria, others are just spurious events. Ideally, one would like to prioritize the exploration of paths that maximize the chance of reaching the target (e.g., finding malware or bugs). Whereas bug-finding strategies tend to prioritize error cases, malware analyses must prioritize the opposite: the maliciousness of a path. Analysts are usually not interested in checking when a malware execution fails, but when it succeeds and infects a system. Once again, the open problem is: what are the criteria to know what is malicious?

Unfortunately, the academic literature cannot still definitely answer what are the best practices for defining maliciousness in the context of the symbolic execution of malware. Each work tends to adopt its own definition (if any), without further validity evaluation. The absence of a clear maliciousness metric is a criticism of many works [16]. Whereas some works evaluate how many additional paths can be found in malware samples [50], they do not evaluate whether the additionally found paths are false positives [62]. This leads to a second large

criticism, that it is the semi-automated nature of the symbolic analysis process [62]. Due to the lack of clear maliciousness criteria, researchers need to manually validate the found paths, such that the development of a completely automated process for malware symbolic analysis is an open problem.

Some works end up adopting ad-hoc maliciousness criteria. For instance, in the analysis of network-based malware, one might define that a path is malicious if it leads to HTTP requests [5]. The major drawback of this approach is that it does not generalize to other types of malware. Some works tried to develop more general approaches such as considering a path as malicious if it presents a given sequence of calls [3]. In this case, however, the approach lacks explainability and generalization beyond manually-defined behaviors.

The reliance on human-defined behaviors (e.g., via taxonomies [39]) is a characteristic of many of the works in the field. Whereas traditional dynamic analysis systems already require humans to select which tools and/or environments analysis will run [32], symbolic analysis takes it to the next level. Therefore, we can overall conclude that it is urgent to develop and evaluate strategies for path prioritization and stop conditions of symbolic execution of malware samples.

API call support: Automating procedures. A key part of any malicious program is the API functions invoked by the samples, that reveal important IoCs and allow understanding how the malware sample interacts with the system. Whereas in the case of fuzzing the adopted strategy was to replace the function's return values, in symbolic execution the typical approach is to symbolize the function return. Due to the symbolization process, the actual execution of the API functions becomes irrelevant, such that one can replace it with a `function summary`, a piece of code that is invoked in the replacement for the originally called function but that does not actually execute, only returns a symbolic variable. Function summaries are used by all investigated papers.

A common challenge for function summarization is to write function summaries for all functions supported by a system and/or linked by malware samples. This number might be of thousands [50], thus it is hard to write them manually. Whereas the decision if a path is malicious under some criteria is not fully automated, the automated writing of function summaries has been reasonably well-described in the literature; in both semi-supervised approaches (e.g., hooking API based on human annotations [3]) and fully automated approaches. In the latter, the approaches crawl function prototypes on the Internet to generate summaries based on them [5, 17].

Symbolic Execution Limitations: Implementations. Developing a symbolic execution engine is complicated and involves making multiple assumptions and project decisions [6]. Many of the assumptions are targeting the general case and/or the specific case of bug-finding. Whereas many of these assumptions might be valid for the malware analysis task, many of them do not hold. For instance, due to the complexity of developing a symbolic executor, many complicated system features are not fully implemented, as they are not used by the majority of traditional applications and/or are not the usual target of symbolic analysis procedures. These often non-implemented features include the support for multi-threading [45], process forking, and Inter-Process Communication (IPC) [17]. Whereas missing these functions might not be an obstacle to many analysis tasks often performed in symbolic executors (e.g., finding bugs), these limitations might be the reason why a malware analysis fails under the symbolic executor, as malware samples are known to exploit the drawbacks of the analysis solutions. In this sense, a benchmark of symbolic executor limitations [90] revealed that less than a third of all tested corner cases were handled by current solutions. Thus, making current symbolic executors more robust is key to streamlining them as malware analysis tools.

Attacks against symbolic execution. As for fuzzing techniques, attackers will react to the adoption of symbolic execution techniques for malware analysis by adopting anti-analysis methods, such as more robust obfuscation techniques to defeat symbolic execution [75]. The symbolic execution technique presents a series of intrinsic limitations that might be exploited by malware samples, such as jumping to symbolic locations, which is proven to be unresolvable by the engines. The limitations might be exploited by attackers without even requiring them to write code to intentionally do that. Attackers might pack their samples with packers designed to purposely cause analysis over tainting when running under symbolic executors [93]. Whereas this type of technique is still

not widely seen in practice, as symbolic execution techniques are not as widespread as other techniques, the foundations for anti-symbolic execution techniques can already be found in the literature. New anti-analysis techniques might even emerge in the next years due to the recent development of using neural networks for control flow obfuscation [55]. In this sense, we believe that attackers are at a significant advantage because even though many obfuscation formulas might be reduced to their original complexity, it takes hours [59], which creates a significant attack opportunity window.

3.4 Concolic Execution

Whereas symbolic execution is powerful, it presents many limitations. For instance, pure symbolic execution approaches are based on static analysis and thus present all the known static analysis limits [61]. As one might expect, these limits are actively exercised by the attackers [71]. The limits of static analysis can only be overcome by dynamic inspection.

This caused a revamp of dynamic analysis approaches. This time, however, the proposed approaches should not only repeat previous fuzzing strategies but also benefit from the lessons of symbolic execution. Therefore, researchers proposed concolic execution, a combination of the `CONCrete` and the `symbOLIC` approaches, benefiting from the strong points of each one of them while mitigating their drawbacks.

A typical concolic execution strategy is to concretely execute an application to reach a deep state and then migrate this state to a symbolic version to locally explore derived paths. This allows one to use concrete executions to bypass hard-to-solve symbolic points and then benefit from symbolic analysis capabilities on the remaining binary regions.

Concolic execution tools are based on taint tracking, a dataflow tracking technique that allows one to monitor the flow of a given byte from one binary point to another. For instance, one might check the flow of function returns from the original call site to a decision point (e.g., a branch). Thus, concolic executors are often implemented via the instrumentation of traditional sandboxes with taint-tracking capabilities. Whereas concolic executors are available for multiple platforms (e.g., Android [89]), the majority of the approach seems to be first developed on x86. A typical implementation decision is to instrument an x86 binary translator (e.g. Intel PIN, DynamoRIO, QEMU) with a flow tracking library (e.g., LibDIFT [46]) to perform instruction-level flow tracking.

The use of concolic execution is interesting for malware analysis as it allows one to overcome the major challenges of symbolic execution exploited by attackers. For instance, it allows one to bypass the effect of packers by concretely executing a binary until it unpacks itself and then symbolically exploring the paths of the unpacked binary [18]. Due to this capability, concolic execution approaches have been achieving greater results than dynamic execution alone [63].

Despite this claimed success, no work in the literature systematically evaluated to which extent concolic execution extends malware analysis capabilities of previous approaches. This type of evaluation is particularly important when we consider the required effort to build a concolic executor, which is significantly greater than other solutions as one needs to build both the concrete and symbolic execution engines.

Concolic Execution Limits. As for previous solutions, concolic execution is not a silver bullet, thus it presents limitations that might eventually be explored by malware authors. In sum, concolic execution inherits all limitations of its underlying techniques. In other words, it presents all drawbacks presented for dynamic analysis plus the new taint tracking ones. The limits of flow tracking techniques are well described in the literature [24] and involve, for instance, race conditions and covert channels. Previous research has already demonstrated how these limits might be used in practice to evade analysis [23]. Whereas concolic execution is not yet the most popular malware analysis technique, we hypothesize that this type of abuse might be future seen in the wild.

3.5 Limits of Previous Evaluations

We reviewed the literature on multipath malware execution from the technical point of view, highlighting development challenges and opportunities. The literature works also present limitations in their evaluations and thus opportunities to build knowledge on multipath malware execution. We here revisit literature findings and highlight open questions.

Table 1 shows a summary of the most relevant works presented in this paper. For each related work (Column 1), it shows the technique used in the work (Column 2); the total number of samples used in the study (Column 3), for the cases in which a large dataset of unfiltered samples is considered; the total number of samples which displayed multipath behavior (Column 4), for those studies triaging samples from a larger dataset; the identified prevalence rate of evasive samples when the study claims that the obtained proportion is representative of the real world (Column 5); the type and/or the age of the malware samples used for the experiments (Column 6); the number of iterations/runs of each malware sample analysis (Column 7); and the overhead and/or the execution time to analyze each sample (Column 8). We left empty entries when the paper does not specify the required information and/or when the calculation does not apply (e.g., studies manually picking multipath samples from the larger dataset without a clear evaluation of their prevalence).

Table 1. Summary of multipath malware execution approaches.

Work	Path Exploration Mode	Total Samples	Multipath Samples	Samples Ratio	Dataset Type/Year	Iterations	Overhead/Time
[4]	Natural Variation	7.6M	1M	13%			
[7]	Sandbox Comparison		10				
[92]	Symbolic Execution	1439			2014		
[60]	Sandbox Variation	300			2007		
[44]	Reference Sandbox	5				7	20 min
[2]	Reference Sandbox	1470	192	13%			
[30]	Sandbox Variation		6				
[48]	Sandbox Comparison	110K	5K	4.5%			
[52]	Sandbox Comparison	1500	400	26.7%			
[47]	Reference Sandbox	2100			2015		
[42]	Reference Sandbox		2				1 hour
[72]	Static + Tainting			3%			
[87]	Forced Execution	951					
[1]	Forced Execution	100					
[64]	Forced Execution		10			26K	47 hours
[27]	Symbolic Execution						78x
[18]	Concolic Execution				MyDoom		30min
[5]	Symbolic Execution		1				
[3]	Symbolic Execution		50				
[50]	Concolic Execution	63K	1K				20 min
[59]	Symbolic Execution		13				300 sec
[93]	Symbolic Execution						12 hours
[63]	Concolic Execution		302				
[89]	Dynamic + Tainting		75				138 sec

The low number of samples limits the approaches' evaluation. A key aspect of a good evaluation is diversity, as it allows one to evaluate multiple aspects and characteristics of a subject. However, most works reported in the literature are still case studies, using only a few samples (e.g., 3 samples [53]). The largest studies published so far only presented limited sets of samples (e.g., a few hundred), without further guarantees of diversity. Although the largest study on the topic presented a million sample evaluations [4], this evaluation consisted of in-the-wild observations and not of a controlled experiment. In this sense, we can consider that only a single work [48] presented a structured evaluation with a reasonably large and thus likely diverse dataset, such that more similar evaluations are required to confirm or not the obtained statistics.

The performance cost of running samples in multiple sandboxes and symbolic executors is a plausible hypothesis for the limited size of the datasets used in most tests. Whereas the cost of each approach described in the literature significantly varies, the average cost of many approaches (noticeably symbolic ones) seems to be around 20 minutes [18, 44, 50], even though it might reach hours [64, 93]. In both cases, this is a significantly higher time than typical single-run sandboxes, that run samples just for a few minutes. This increased requirement for CPU time might limit the scope of many studies in the field, which opt to run a sample multiple times to discover hidden paths in comparison to traditional studies that run more samples only once in the same time slot.
The lack of prevalence data limits the characterization of the occurrence of multipath malware in the wild. This information would be key to defenders knowing to which extent they might be losing information about malware samples by looking at individual runs and limited sets of IoCs. Unfortunately, most works do not work with representative datasets, but only limited and even purposely biased collections for the effect of testing the proposed approaches [50]. The few studies evaluating multipath malware prevalence presented results ranging from 3% [72] to 26% [52]. This high variation highlights the need for better analyses of these results.
The aged samples is also a limitation of most experiments found in the literature. Finding malware samples with relevant multipath execution is challenging, thus many researchers opt to use known samples (e.g., MyDoom [18]) as ground truth for experiments. Whereas this is scientifically valid, this limits our understanding of the current scenario, as these samples were collected many years ago (e.g., 2007 [60], 2014 [92]). Thus, updating the results for recent threats is key to enriching our knowledge about multipath malware.
The lack of tools comparisons complicates the establishment of a more robust verdict on which one of the multipath execution approaches presented in this paper is the most effective in finding malicious paths. Most of the presented works are focused on evaluating a proposed solution rather than comparing multiple solutions. Understanding the tool's cost-benefit is essential to foster the adoption of multipath execution solutions because this type of solution might be expensive (both in acquisition cost, as well as in development and deploying costs). Thus, defenders must be able to properly evaluate which tool best fits their budgets and usage scenarios.

4 EMPIRICAL ANALYSIS

We previously showed that the studies described in the literature are limited in multiple aspects. Noticeably they lack data from real-world scenarios in their evaluation, which limits the drawing of their application landscape. To bridge this gap, we conducted multiple experiments that apply the techniques described in this paper to a dataset of malware samples collected in the wild. We following present our findings to draw a landscape of the application of the discussed solutions in practice.

4.1 Methodology

Threat Model. Although multi-path analysis techniques have been proposed for many environments, such as Javascript [49] and Mobile [95], in this work, we limit our analyses to binary code for the Windows platform, the most targeted by the attackers and the environment for which most tools described in the literature were developed for. We limit our scope to cover userland threats, as no multi-path kernel rootkit case was found in the literature review. Our investigation focuses only on the binary code present in the binary (e.g., text sections) and does not cover the library's internal code, as they present their own drawbacks [72] and we aim to spot attackers' intentional constructions.
Target Scenario. The application scenario is an analyst running malware in a typical sandbox solution. The sandbox goal is to produce the most accurate trace possible to allow (i) the analyst to identify malicious behaviors in it or (ii) automated solutions to be applied to it to classify the sample.
Research Goal. We reinforce that our goal is not to evaluate which is the best tracing solution in absolute. We understand that all existing techniques should still be more developed to be practical. Our goal is to characterize

them to understand their readiness level and what should be developed to move them forward. We do this from the perspective of an analyst using a sandbox solution powered by each one of the investigated techniques.

Algorithm 1: Alignment Algorithm.

```
   Data: Trace1, Trace2
   Result: Deviation or Not
   /* Start with no differences                                    */
1  diffs_t1[] = diffs_t2[] = [];
   /* Align Sequences                                              */
2  Seq1, Seq2 = LCS(Trace1,Trace2);
   /* Get first line each sequence                                 */
3  t1 = Seq1[0]; t2 = Seq2[0];
   /* While having data                                            */
4  while has_data(t1) or has_data(t2 do
       /* Advance while matching                                   */
5      while t1==t2 do
6          next(t1); next(t2);
       /* T1 mismatch, add to diff list                            */
7      while missing(t1) do
8          diffs_t1.append(t1); next(t1);
       /* T2 mismatch, add to diff list                            */
9      while missing(t2) do
10         diffs_t2.append(t2); next(t2);
   /* For all mismatches                                           */
11 for block in diffs_t1 do
       /* Blocks were out of order                                 */
12     if block in diffs_t2 then
           /* Not an actual mismatch                               */
13         diffs_t1.remove(block);
14         diffs_t2.remove(block);
   /* If remaining blocks                                          */
15 if diffs_t1 != diffs_t2 then
16     return "Deviation";
   /* Else, empty list, no deviation                               */
17 return "No Deviation";
```

Fuzzer implementation. There is a great availability of fuzzers for vulnerability discovery, but they are not suitable for malware analysis, because they do not allow, for instance, changing function returns. Thus, we implemented our own fuzzer (and tracer) for the sake of evaluation. Our fuzzer is based on Intel PIN [54] and it can identify when it is running binary and library code via PIN's native libraries. This is key to computing coverage correctly, as we do not want to mix library code with binary code in the measurements. The fuzzer hooks library functions to log their arguments, which allows for identifying distinct paths, The hooks are also used to adjust the return values, for fuzzing purposes. We hooked the same API functions hooked by the Cuckoo sandbox [73], as it is a popular sandbox, and many malware analysts also hook these functions. In addition to changing function returns, we added the fuzzer the ability to take arbitrary branch directions, which is key for

implementing forced execution approaches. We implemented this feature by collecting the target branch address and directly jumping to it, despite the actual comparison evaluation result.

Symbolic execution strategy. For pure symbolic execution, we relied on Angr. We wrote function summaries to collect concrete arguments for logging and modified the function summaries to return a symbolic value that would be used by the malware to further decide if branches would be taken or not. To symbolize functions at scale, we developed a crawler that collects function prototypes on the Internet and automatically generates the summaries, a strategy proposed by previous research [17]. Since Angr does not provide a native solution for concolic execution on Windows, we relied on Triton and S2E for this step. Whereas, in the first case, we were able to directly symbolize function returns via the Triton API, in the latter, we relied on function hooks to inject symbolic values in the function return addresses, as also proposed by previous research [40]. Regardless of the used solution, before running symbolic execution in batch, we tuned the engines for the best parameters, as suggested by previous work [76].

Alignment algorithm. A key aspect of claiming the discovery of a new path is to be able to compare two traces and recognize significant differences. For our experiments, we implemented an algorithm that, given two call traces, it tries to align them. For each misalignment it identifies, it reports the function missing in one of the traces. If the traces are aligned, the function pairs are checked to report if they have different arguments or return values. This flexibility allows us to report divergences at different granularities (entire functions, different returns, and different arguments).

The algorithm also checks for pseudo-divergences, i.e., when two blocks are not aligned due to temporal issues (e.g., context switches) but they are equivalent. This is possible because our algorithm performs an exact match, comparing all alignment possibilities. Whereas it is very slow (quadratic) to be performed instruction-wise, it is feasible to perform it at the API trace level. Due to this implementation choice, our analyses do not present False Positives. The time taken to compute the divergence is not considered in our experiments.

Algorithm 1 presents a high-level (but still detailed) view of the inner workings of our base alignment algorithm. It receives two traces as input. If more traces are required to be compared, they are compared two by two in consecutive operations. The algorithm starts performing the Longest Common Subsequence (LCS) [10] matching to align the two traces provided as input (line 2). When the tested tools output the results in log/text files (e.g., our custom fuzzer), we directly compare LCS via the `GNU diff` utility [35]. When the results are presented in custom formats, we parse them and compute LCS via a modified Python library [56]. In both cases, the outcome is the alignment of the two sequences (Seq1 and Seq2). We further iterate over these sequences (line 4) to identify the blocks that are not aligned (lines 7 and 9). These blocks are stored for future comparison (lines 8 and 10). Once all blocks are identified, we verify if there are common unaligned blocks between the two traces (line 11). When it happens (line 12), it means that the unalignment occurred due to timing issues (e.g., thread scheduling), and not due to actual deviation. The algorithm finally checks if there are remaining deviating blocks. If so (line 15), it reports the traces as deviating. It reports the traces as compatible otherwise (line 17).

Dataset. This study's goal is to observe the impact of multiple execution paths in a real scenario. For such, we considered a dataset representative of real user infections. Our experiments were conducted on a dataset of 21K samples collected by a CSIRT team directly from infected users' machines over 8 years (2012-2020). This dataset is composed of malicious EXE files from multiple families and it was fully characterized in a previous study [12]. Samples' maliciousness was corroborated by Virustotal scans. The dataset was not filtered in any way, thus presenting all the characteristics (and biases) of the original collection scenario from the user machines, thus reflecting the real-world setting. The dataset is composed of more than 100 families, with 70% of the samples presenting the downloader, exfiltrator, and/or banking behaviors. In total, 66% of all samples are singletons, and 33% are variants. The samples were considered *"as-is"*, without disarming, to reflect the sample the analyst would submit to the sandbox. In total, 50% of samples were packed, thus affecting symbolic execution approaches, as in real-world settings. The full dataset characterization is presented in [12].

Evaluation metrics. We evaluated execution paths according to two different metrics: API traces and coverage measures. We clarify in the text when each measurement type is used. Coverage is measured in terms of block and edge coverage. The reader can assume block coverage when not stated otherwise. The coverage computation is performed over the complete binary disassembly. Since most malware binaries are stripped, we cannot guarantee that binary data is separated from instructions. We ensured the correctness of our coverage measurement by running known applications under our solutions and WinAFL and comparing the obtained results.
Comparison fairness. Our research goal is **not** to compare approaches to tell which one is the best for malware tracing (this is left as future work). Instead, our goal is to **characterize** the occurrence in practice of the effects theoretically described in the literature. Thus, the approaches implemented in our solutions are inspired by previous works' development, but they are **not** re-implementations of the previous works. It would not scale considering the large amount of covered tools. Rather, we present under-approximations–i.e., we do not guarantee to provide the same features as the related works, but we implemented solutions that employ the same classes of techniques and thus achieve comparable results to a significant extent. We believe this is a reasonable decision since the differences between the classes of solutions are significant enough to be highlighted by the approximations. In turn, exact re-implementations would only benefit intra-class comparisons (out of scope).
Watchdogs and Timeouts. During distinct runs, a distinct number of instructions might be executed due to natural reasons. To provide a fair comparison among traces of multiple sizes, we implemented a watchdog that limits the number of traced instructions (detailed in Section 4.3). When performing forced execution experiments, we, unfortunately, do not have infinite resources to explore all possible paths. We defined a limit of 1024 distinct runs as a reasonable amount. In the worst case, it enables the full exploration of distinct paths at a depth of 10. In the average case, it allows delving deep into the binary as we count only branches related to function invocations. We also set a timeout for the symbolic execution to 10 hours. In total, the set of 21K malware samples resulted in more than 21M sandbox executions over 1 year.

4.2 Natural Variation

We started our investigations by checking how much variation in malware executed paths is natural–i.e., it happens naturally, without the need for external stimulation, and it is not necessarily triggered on purpose by the malware samples. This information is key to understanding to which extent multipath execution is a problem. It is also key to establish ground truth for future comparisons, as only by understanding how much variation is natural we can understand if a given technique found more paths than others. To evaluate natural path variations, we sequentially ran all samples in a sandbox 10 times each for 100M binary instructions (see Section 4.3 for details) and compared their function call traces for significant differences.
How much variation is natural? Figure 2 shows the malware sample's distribution according to the number of distinct execution paths they presented during the 10 runs. We notice that ≈38% of all samples only present a single execution path. We consider that the existence of a single and/or preferential execution path is supported by the statistical significance obtained from the 10 runs, in which the samples had multiple opportunities to display natural variations.

Most samples (≈62%) presented at least one diverging path during the 10 runs, thus we counted them as samples that present natural variations. Since the samples in our dataset have no previously known multipath characteristics, we believe we can generalize this result to many malware datasets. Thus, the occurrence of natural variations might be a very frequent phenomenon, even though it is rarely acknowledged in the literature reports. Based on that, we advise that researchers should not overlook variations in most malware analyses.

Half (50%) of these samples (31% of the total) presented an intermediate number of distinct paths (2-9), which means that some of the executed paths repeated themselves among distinct runs. This result implies that another half of these samples is significantly affected by natural variation effects, as the maximum number of 10 distinct

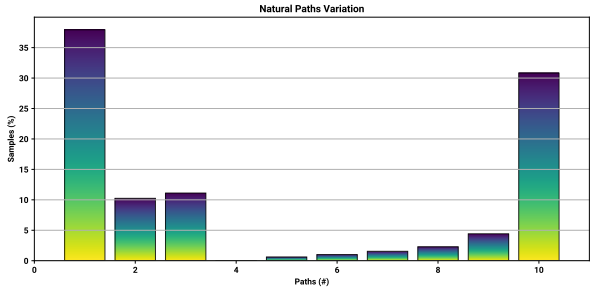

Fig. 2. **Natural Path Variation.** The majority of samples present naturally executed path variations.

exhibited paths is bounded by the number of times we repeated their executions and not necessarily by the sample's characteristics. We understand that this result is another piece of significant evidence that the effects of path variations should not be neglected.

What are the causes for trace differences? We found two reasons for the alignment algorithm claiming that two traces were different: (i) actual differences in the trace functions (e.g., different function arguments); and (ii) differences in the order in which the functions are present in the trace. Whereas the first happens due to how the samples are implemented, subject to randomness, the latter happens mostly because of OS scheduling, such that the order in which the threads are scheduled changes the order in which the functions appear in the trace. Thus, whereas the first case is the actual study subject of this paper (causes are below detailed), we also measured the latter to eliminate this effect from overall statistics. In our experiments, 3.37% of all samples were affected by trace divergences due to OS scheduling. We do not account for it in the divergence statistics.

Where does variation happen? Whereas most variations occurred directly in the launched malware process, we notice that for 17% of the diverging samples the variation occurred in a child process. Although this case is not the majority, symbolic executors [17] and other solutions that do not track child process are missing significant variation cases.

What was found in the diverging paths? Despite the number of distinct paths naturally observed during the multiple runs, no significant deviation was observed. In a few runs, the diverging samples presented distinct IoCs (1% of them presented distinct IP addresses and 3% presented distinct DNS requests) while implementing the same malicious behaviors in the same order. Based on that, we conclude that multiple execution runs can reveal IoCs related to the sample's random-based code constructions. However, we did not observe any case in which the further-obtained traces presented a particularly different malicious goal than the first-obtained ones (i.e., no evasive behavior). Therefore, researchers need more focused techniques than simple sample re-runs to uncover suspicious behaviors.

4.3 Fuzzing

We proceeded with our evaluation by checking the effect of fuzzing API function returns of an entire dataset of malware samples. We fuzzed the API returns as described in the Methodology section (Section 4.1) and varied the amount of fuzzed function calls to identify to which extent the samples are affected by these changes.

Does fuzzing malware discover more execution paths? Figure 3 shows the number of new unique paths (in terms of function call traces) that we identified on average for each sample in the dataset when fuzzing them with a distinct amount of external randomness (i.e., with distinct probabilities of a given API function return being modified in that run). We ran each sample 300 times for each randomness value for up to 100M main binary instructions (see full justification in the experiments below). We notice that when no external randomness is

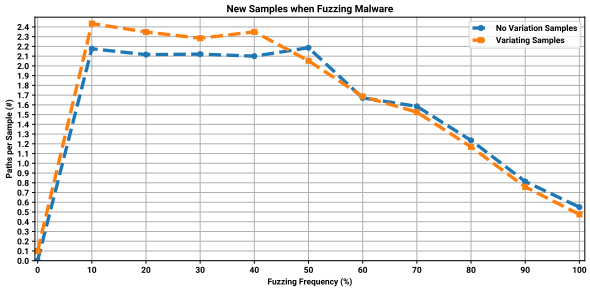

Fig. 3. **Discovered paths when fuzzing.** There is a trade-off between fuzzing frequency and the number of newly discovered paths.

applied, the average number of newly-discovered paths per sample is low, as only natural randomness is observed and we attempted to discard these cases the most possible from our statistics based on the trace results from the previous experiment. When we start applying external randomness to the samples, i.e., we start changing API returns, more unique paths are discovered for each sample, for all amounts of randomness added to them. Thus, we conclude that fuzzing malware does result in the discovery of new paths.

Which samples are affected by fuzzing? Figure 3 shows the average number of newly discovered paths both for the samples that presented natural variation and for the samples that did not present variations in the previous experiment (thus they start growing from the absolute zero). Although the average number of newly discovered paths is greater for the samples that naturally tend to present variations, as one could hypothesize, we notice that the samples that originally did not present variations were also affected by the fuzzing effect. This shows that fuzzing malware is an interesting strategy to uncover hidden malware execution paths for most samples.

How much fuzzing is required? Figure 3 also shows results for different fuzzing rates. For each rate, each function invoked by each malware sample has that given chance of having its function return modified by the fuzzer. The rationale behind that is that after multiple runs, distinct function returns are modified each time such that distinct paths are executed. Whereas one might hypothesize that the greater the probability of modifying a function return the better, this is not true. In practice, there is an ideal amount of fuzzing one should apply to a binary, such that the binary takes distinct execution paths at each run. We notice the greater number of average newly-discovered paths happens when all functions have a 20% chance of having their return values modified. Modifying all function returns is not a good strategy because it always guides the execution via the same paths, such as via the same error cases. We notice this effect in practice in our dataset by observing that when all function returns (100%) are modified, the number of newly discovered paths is minimal.

Are function call traces enough? The results presented so far allowed us to show that fuzzing malware leads to the discovery of more API calls, but this does not exhaust the path exploration subject. In some cases, distinct paths might lead to the same APIs in distinct ways, and discovering these cases might be relevant for some analysis procedures (e.g., malware resilience evaluation). Thus, we need to evaluate also the taken paths, which is performed via coverage analysis.

Does trace size matter? Before evaluating code coverage, we need to evaluate first which trace size to consider to do so. The more instructions we let a sample run, the larger the binary part that tends to be executed. In our experiments, we tried multiple configurations to limit the trace size. Table 2 shows block and edge coverage for the malware samples when no fuzzing is applied and the trace size is limited to multiple values. Although the coverage initially grows along with the trace size, it stops growing after 100M main binary's instructions, thus

Table 2. **Natural Malware Coverage vs. Trace size.** The size of the trace affects the executed malware portion..

Instructions	1M	10M	100M	1B
Block	5.65%	7.70%	9.52%	9.85%
Edge	5.01%	7.32%	9.66%	9.76%

we used this value as the trace size limit for all experiments presented in this paper. Notice that we do not count library instructions on the coverage value, but only instructions belonging to the binary image.

What is the typical malware coverage value? As far as we know, this information is not available in the literature, which motivates our investigation. Most malware reports do not explicit which portions of the binary are exercised. Even though these reports present rich information about malware behavior, it is important to have a sense of how much of the whole malware sample we really understood. Table 2 shows that the natural malware coverage during a sandbox run is low (≈10%) on average. This is a significant contrast to offensive security reports in the literature, whose fuzzing results might easily reach 90%. However, we highlight that our results are the average coverage among all samples in the dataset, whereas typical fuzzing works normally present individual samples' coverage. Whereas individual samples in our dataset have also presented high coverage, we highlight that most samples in the dataset execute a small set of instructions among all possible instructions in their binaries. This does not mean that the malware samples are not active, but that most of their execution occurs in the libraries rather than in the main binaries, our focus in this work.

Why are there differences between block and edge coverage? There are many metrics to evaluate coverage. Two popular ones are block and edge coverage. The rationale behind them is that block coverage evaluates the set of reached instructions whereas edge coverage evaluates the multiple ways to reach the blocks. It is plausible for these two metrics to differ, as there might be multiple ways (edges) to reach the same blocks. In our tests, however, the results for both metrics were very similar, thus indicating that most malware samples have a more linear execution flow–although there are exceptions for individual samples–with most edges leading to a single block, such that the metrics converge.

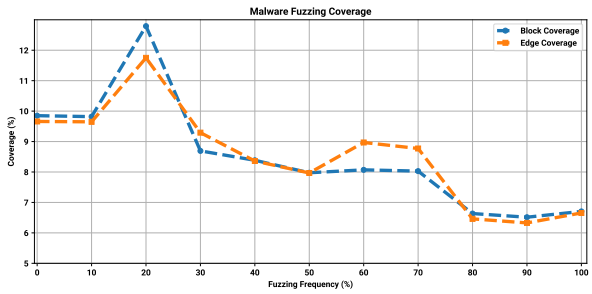

Fig. 4. **Fuzzing Coverage.** There is an ideal fuzzing frequency to reach maximum coverage.

Does fuzzing increase malware code coverage? Figure 4 shows block and edge coverage for the same previously reported fuzzer runs. We notice that fuzzing can increase malware coverage, in terms of both blocks and edges. However, as for the previous experiment, there is an ideal amount of fuzzing, as the coverage increases only in the 10-30% interval, with a peak in 20%, and decreases after it.

So should one use coverage to measure malware paths instead? Whereas this result shows that coverage is an important indicator for malware fuzzing, it does not say that coverage should be the only evaluation metric.

Instead, code coverage should be evaluated in conjunction with API traces, because although coverage decreases when too much fuzzing is applied, previous results showed that these same paths also led to the emergence of new functions in the traces.

What's the fuzzing effect on block and edge coverage? The fuzzing results presented in Figure 4 overall allow us to draw the same conclusions from the natural variation experiment, with block and edge coverages following each other, thus suggesting that most samples in the malware dataset have a more linear execution flow. More than that, we notice that the edge coverage tends to be lower than block coverage in most cases, which suggests that new blocks are discovered first than new edges. however, this is not true for the whole fuzzing range. In the 60-70% interval, edge coverage is greater than block coverage, which suggests that in some cases, a fuzzer can discover new ways to reach the same set of blocks.

4.4 Forced Execution

We previously showed that fuzzing API function returns finds more paths than the standard sample execution, but fuzzing depends on luck to find the paths, which limits the number of paths one can find. We here show results for running a similar experiment–with the same samples and in the same sandbox settings–but now forcing execution states to branch at every path found during the main binary traversal.

Table 3. **Coverage Increase.** Forced execution can significantly increase code coverage.

Increase (%)	Samples (%)	Increase (%)	Samples (%)
$0 \leq$ Increase $<$ 10	18,18%	$100 \leq$ Increase $<$ 1000	35,06%
$10 \leq$ Increase $<$ 100	40,26%	Increase $\geq$ 1000	6,49%

Does forced execution find more paths in all samples? A major motivation for using forced execution is that, theoretically, it can deterministically find all hidden paths in the malware samples. In practice, the discovery of meaningful paths completely depends on the internal malware binaries' structure. We noticed that the coverage tends to increase in batches–when key decision branches are reached–thus the coverage increases cluster in blocks. To simplify visualization, we here present the results in terms of the clusters of coverage increase. Table 3 shows the distribution of coverage increase ranges for all samples in the dataset in comparison with the standard sandbox execution. In our tests, ≈18% of all binaries did not significantly increase (>10%) their coverage even after multiple runs, which shows that a minority but still a relevant number of samples do not have relevant secondary paths.

Does forced execution find more execution paths than fuzzing? Although not all samples present hidden paths, a significant reason to adopt forced execution rather than fuzzing is if it can find more paths in the samples that present at least one alternate execution path. The results in Table 3 show that the code coverage significantly increased for the majority of samples (≈82%). For more than 40% of the samples, the coverage obtained via forced execution is more than double that in the original execution. For some samples (≈6%), it is more than 1000 times greater, which shows that the original execution was very limited, which is the case for many samples in the dataset. On average, forced execution resulted in a 3.17 greater coverage than traditional fuzzing.

Does forced execution equally affect all samples? Once again, even though forced execution inverts the branch direction of all samples, the number of new paths discovered in each sample depends on the sample's internal structures. Figure 5 shows the coverage growing after multiple runs (exhibition limited to 300 for readability) for 5 the real samples presented in Table 4 that were selected from our dataset to didactically illustrate the phenomenon. The coverage has grown for all samples, but more in some samples than in others. Some samples even reached high coverage values (60%), even though the average coverage in the dataset is low, thus

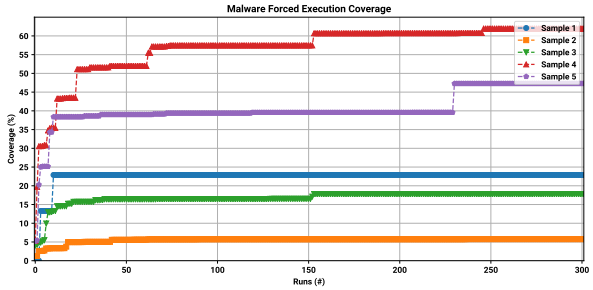

Fig. 5. **Forced Execution Coverage.** Distinct samples are affected in different proportions.

Table 4. **List of Malware Samples** used in the experiments shown in Section 4.4.

Sample ID	Sample Hash (MD5)
Sample 1	70f41afd6dea5ae79f1fc8316c62720f
Sample 2	0126396b1598ac3855b5e9318188e627
Sample 3	be831593163cbe4651dd46900b617ffd
Sample 4	64d90fe866e8580546a538e7b3d40bfc
Sample 5	eadb4a67f612b287892615526a6b4bd9

showing that there are samples with exceptional constructions in them. The pace of the coverage growth in each sample is different: the coverage grows faster for some same than for others. Moreover, whereas some samples quickly reached their coverage limit, other samples took a long time, thus indicating that the divergence occurred in deep program states.

Is waiting for the execution of all paths a viable strategy? Figure 5 also illustrates how forced execution might take a long time to discover new program states: Sample 3 takes more than 90 consecutive runs (60-150) to grow its coverage again; Sample 4 was still growing after 240 runs. Since forced execution must explore all branch directions, new states might be discovered only after multiple runs, thus execution time is a significant limitation. Ideally, we would like to have faster mechanisms to identify new paths.

4.5 Coverage-Guided Fuzzing

A typical way to speed up path search is to give a greater focus on the coverage metric by prioritizing the exploration of paths that tend to present greater coverage. This strategy is often implemented as a DFS search based on the results of previous explorations, such that the exploration of the most promising path in terms of coverage is always prioritized. We repeated this strategy in our malware experiments by keeping a max-heap of the path of greatest coverage achieved until a given branch and then resuming the malware execution from there in 2 runs: one taking and the other not taking the next branch. The new coverage is computed for both paths and they are inserted into the heap. It repeats until the stop criteria.

Does a coverage-guided strategy help coverage to grow faster? The coverage-guided strategy is clearly effective in growing the sample's coverage: It took less than 100 runs to make the Sample 4 from Figure 5 reach the same coverage level reached in 250 runs via the forced execution approach. On average, it reduced in 15% for each sample the number of runs to reach the same coverage as via forced execution.

Is growing the coverage enough? Figure 6 shows an excerpt of Sample 4's initial runs. The left y-axis shows the achieved code coverage and the right y-axis shows the number of found paths API-wise. We notice that

coverage significantly increased (almost doubled) in multiple opportunities (runs 6-10 and 17-21) without being followed by a growth in the number of API paths. It happens because the coverage increased by exploring error cases, which all led to the execution termination via the same well-known API sequence. This result shows that increasing code coverage does not guarantee that more relevant malicious actions will be found. If we keep exploring these states, the exploration will be too focused on error cases and not on actual malicious behaviors, such that this result reinforces the need for malware-focused path prioritization strategies.

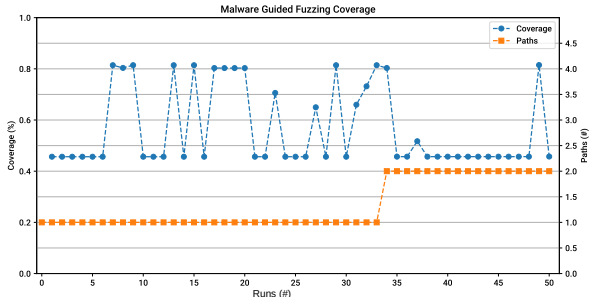

Fig. 6. **Coverage-guided fuzzing.** Fast coverage growth, but no guarantees that key paths will be explored.

After the error cases, the exploration resumes from the previous coverage levels (runs 10 and 21) because the distinct branch directions must be explored from there. Note that there is a coverage increase followed by a corresponding new path in the 30-35 interval. The graph clearly shows that the increase results from the (taken/not taken) decisions of multiple branches, which indicates it is a complex decision. While the coverage-guided approach could identify the complex construction, the time of its discovery is still based on luck, as in pure fuzzers, since this type of solution can't tell apart the error cases. Thus, besides a solution that prioritizes paths, we need a solution that precisely identifies complex decision conditions, the promise of symbolic executors.

4.6 Symbolic Execution

We attempted to analyze all malware samples from previous experiments using pure symbolic execution. However, it presented many drawbacks to handling packed samples (even after previous unpacking), which limited the analysis of 52% of them. In total, 88.5% of all samples did not produce significant analysis results; 10% produced a single execution path; and 1.5% produced two or more paths. Thus, we concluded that pure symbolic execution of malware is theoretically possible but impractical for large, diversified datasets.

How is the symbolic execution performance characterized? Symbolically executing a code piece is slower than its native execution, since it requires interpretation. In our experiments, the difference in throughput between the two scenarios was two orders of magnitude. Whereas native code executed at an average rate (considering all samples) of 21M instructions per second, the code being symbolically analyzed executed at a rate of 226K instructions per second. If we alleviate the DSE load by replacing the libraries' code with function summaries, the throughput increases to 253K ins/s, which is faster but still two orders of magnitude slower.

Symbolic execution takes longer than other approaches to produce meaningful results. Although fuzzing requires multiple runs to produce comprehensive results, each run produces a potentially informative path for the analysts. Symbolic execution, in turn, explores multiple paths in parallel, such that although broader results (more paths) are presented in the end, they take longer to be produced. Changing the search strategy to DFS tends to produce the first result faster, as meaningful information for malware analysis (e.g., network calls) is often hidden in deep paths. However, this change also tends to make the DSE stay focused on the same binary

regions, reducing diversity. In our tests, it takes, on average, 7 hours for the DSE to find network-related activity in the successfully analyzed samples. This highlights the need for better malware path prioritization strategies.

4.7 Concolic Execution

Pure symbolic execution was not able to analyze most malware samples due to known drawbacks of static analysis and code interpretation, such as the inability to handle packers. We attempted to mitigate this drawback by letting samples unpack in a concolic execution environment and transferring the context to the symbolic executor. The results are following discussed below.

Is concolic execution more effective than fuzzing and forced execution? Concolic execution found more paths than simple and guided fuzzing, which was expected because the symbolic component of concolic execution deterministically finds paths. In this sense, concolic execution found the same paths as the forced execution approaches, which was also expected because both deterministically explore all paths. It did not find more paths than forced execution as we symbolized only function returns instead of arguments–which is left as future work.

Is concolic execution more efficient than guided fuzzing and forced execution? In our experiments, the guided fuzzing approach was the first to grow its coverage (usually in less than 50% of the time spent for concolic and forced executions), but it comes at the cost of discovering a limited number of paths. Forced execution requires multiple runs (at least one per new path) to identify all paths, whereas concolic execution can identify multiple (on average 18) distinct paths per run. However, running the forced execution approach at natural execution speed is faster than the symbolic execution step speed (5 times, on average, in our test). As a rule of thumb, it is advantageous to run concolic execution rather than forced execution the deeper the symbolic portion is in the program trace because most of the time, both concolic and forced execution will be running at native execution speed.

How well does concolic handle armored samples? The major challenge for handling packed samples is to find the right place to start symbolizing functions. If functions are symbolized prematurely, still in the unpack step, the analysis is taken back to the pure symbolic execution state. In our experiments, we attempt to always start symbolizing function after the unpacker symbolic jump. This strategy succeeded in analyzing all samples packed with UPX (50% of packed samples, 33% of total), but failed for other packers.

Are there limitations to the practical application of concolic execution? In our tests, we took turns between 3 distinct DSEs, as no concolic execution engine could successfully run all samples. A major reason for that is the limited implementation of function summaries: Less than 1% of all Windows API functions invoked by the samples were originally supported by the engines. We mitigated this issue by automatically writing summaries that only return a symbolic value, which was enough to address the majority of the cases (67% of runs). For the remaining runs (33%), the execution finished because of an inconsistent function's internal state.

4.8 Summarizing the Findings

Once we showcased the effects that multipath tracing tools are subject to, we here consolidated these results to summarize how these effects articulate among themselves. We do it from three dimensions: (i) from the analysis success perspective–if the tools were able to analyze the samples; (ii) from the analysis evolution perspective–how the tools discover new paths over time; and (iii) from the analysis performance perspective–how much time/resources the tools required.

Table 5 summarizes the results for the analysis success metrics. When we consider successful executions (1st column) as a matric, in the sense of finishing an analysis, regardless of finding deviations or not, we notice that concrete execution approaches were able to inspect all samples. In turn, symbolic executors were not able to analyze most samples, not even via concolic execution, since the symbolic step was revealed to be the main

Table 5. **Analysis Results.** Analysis summary broken down by technique.

Technique	Execution Success	1+ Paths Found	Variation in Child Processes
Sandbox	100%	62%	17%
Fuzzing	100%	100%	17%
Forced	100%	100%	17%
Coverage-Guided	100%	100%	17%
Symbolic	10%	1.50%	0%
Concolic	10%	1.50%	0%

bottleneck. These approaches successfully ran an order of magnitude fewer samples, which highlights the need for more developments in this solution class to make it practical.

When we consider the discovery of new paths (2nd column), even though not necessarily evasive ones, we notice that repeated sandbox runs are enough to generate different traces in most samples, thus showing an effect that cannot be neglected. However, to actually cover all samples, external stimulation is required. All other concrete approaches found distinct paths (even though an error case) in all other samples, thus demonstrating that external stimulation is key to finding new paths. Symbolic executors once again presented limited performance, not being able to find multiple paths even in all of the samples that they were able to analyze, which shows that path exploration strategies must be more developed to be practical.

To understand where developments should be focused, it is key to understand where the deviations happen (3rd column). We notice that 17% of all diverging paths are located in child processes of the malware initially analyzed. Whereas all concrete executors were able to identify it by following the processes, symbolic executors missed these cases as these solutions do not model the process creation step, being this a significant development required to make malware analysis practical.

Table 6. **Analysis Evolution.** Unique Paths are broken down by technique and the number of iterations.

Technique	10 runs	250 runs	300 runs	1024 runs
Sandbox	1	N/A	N/A	N/A
Fuzzing	N/A	2.1	2.4	N/A
Forced	N/A	2	2.1	7.6
Coverage-Guided	N/A	2.4	2.4	N/A
Symbolic	N/A	N/A	N/A	N/A
Concolic	N/A	N/A	N/A	N/A

Table 6 shows the evolution of the analysis procedures in terms of the unique paths per sample found with the help of each technique. We consider unique paths as a path presenting at least one distinct IoC from the reference (first) run. The rationale for the evaluation is that if two solutions identify the same paths, it is preferable to use the solution that does it early, to speed up incident response. To evaluate that, we broke down the number of identified paths in iteration ranges.

Although repetitive traditional sandbox execution allows discovering differences in the paths (e.g., order of function invocations), as shown in Table 5, these paths tend to present always the same IoCs, thus not leading to new infection paths. The previously observed path differences can be mostly credited to the natural variations, explained by thread scheduling issues. More iterations do not significantly help to reveal more IoCs in the average case, although a few samples were individually affected, but not in enough number to be statistically significant.

Compared to natural sandbox runs, all concrete executors presented more unique paths per sample, thus showing that external stimulation not only generally affects more samples, but also actually discovers new IoCs. Forced execution was the approach that discovered more unique paths in total, but it was also the one that required more iterations to do so. Among the fuzzing approaches, coverage-guided and traditional fuzzers discovered the same number of unique paths in the end, but the coverage-guided fuzzer required 50 fewer iterations to do so.

Symbolic executors were also not considered statistically significant, as they analyzed only a minor part of all samples. Even though, for the cases where symbolic executors returned results, the results were compatible with the forced execution results for the same samples.

Table 7. **Analysis Performance.** Analysis summary broken down by technique.

Technique	Execution Speed	Paths per Run	Time per Sample
Sandbox	1x	1	5 min/path
Fuzzing	1x	1	5 min/path
Forced	1x	1	5 min/path
Coverage-Guided	1x	1	5 min/path
Symbolic	100x	18	7 hours
Concolic	20x	18	84 min

Table 7 compares the different approaches regarding their execution performance (runtime) requirements. The rationale behind the evaluation is once again to prefer the fastest solution when the results are equivalent. The first aspect to be observed is the slowdown imposed by each technique in comparison to the natural execution (1st column). While all concrete executors run at native speed (1x), symbolic execution runs at two orders of magnitude slower (100x). For concolic executors, whereas their concrete execution step decreases the overhead 5 times in comparison to pure symbolic execution, their subsequent symbolic execution step is still a significant bottleneck for the process, causing a 20x slowdown.

The difference in the slowdown should be observed in the light of how many runs at these speeds are required to obtain the reported results (2nd Column). Whereas fuzzers run fast on each path, they require multiple runs to cover the multiple paths. In turn, although symbolic executors are slower, it is key to remember that they always explore all paths in a single run, thus always taking the same analysis time. Therefore, a fair comparison should consider the total time spent in all analysis paths (Column 3). In this sense, whereas pure symbolic executors are really slower, concolic executors would be comparable with concrete executors, because although they impose a 20x slowdown, they also impose an 18x gain in found paths by run. The major limitation for the practical deployment of symbolic executors is still the bottlenecks of the symbolic execution part, which should be prioritized in future developments of symbolic execution for malware analysis.

5 IMPACT ON DETECTION APPLICATIONS

After we analyzed the path exploration capabilities of the multiple tracing approaches, we investigated how the ability of retrieving multiple paths for malware samples influences the analysis procedures. To that, we first demonstrate the overall impact of hidden paths in the creation of malware landscapes. Further, we demonstrate how hidden paths affect detection solutions.

5.1 Divergence Analysis of a real-world dataset

An immediate application of multipath malware execution is to draw a more accurate view of a threat scenario. Whereas previous works drew a landscape of specific scenarios [12, 85], how much information are they missing by not considering multiple executions? Whereas previous work pointed to the specific evasion causes in armored

samples, these studies are based on evasive sample selection and not observation in-the-wild (Section 3.5), which does not allow us to have information about the prevalence of such cases. This information is key to inform the design of future solutions to handle multipath execution. Thus, we here present a multipath analysis of the dataset initially characterized by [12]. We run the same samples in all multipath executors and report the reasons why the malware samples diverged in their executed paths, regardless of the technique that found these paths. Our goal is to **identify and give numbers** to the prevalence of diversion causes to provide researchers with a base for comparison for their future developments.

The total number of multipath samples identified in the studied dataset in our tests is 3%, when considering the unique results of all analysis solutions that presented at least one different Indicator of Compromise (IoC) in the multiple traces. While there is no ground truth for the in-the-wild samples to verify the results, we believe this is the best inference on the real-world prevalence of evasive samples to date. This value is in line with the reports of one of the previous works reported in the literature (Section 3.5), but with a larger confidence margin due to the increased dataset size. We following discuss the distinct diverging causes we found.

Divergence due to evasion attempts. A traditional reason for malware samples presenting distinct execution paths is to evade detection. Whereas this work is not particularly focused on surveying evasion cases, multipath malware execution ends up naturally revealing some strategies. Thus, we analyze the evasive scenario to provide some quantitative and qualitative feedback about evasive malware. A popular way to evade detection is to check for the presence of a debugger (e.g., via `IsDebuggerPresent`). We identified that checking deviations caused by this function is hard, especially because in many cases this function is used internally to the libraries linked to the malware samples and also because this function is often invoked at process startup by the programming language runtimes (e.g., CRT libraries). Excluding these cases, the use of the debugger check function accounts for 19% of all samples. From these, 53% were affected by the intentional modification of the function return, which shows that almost half of the samples do not simply quit execution in the presence of a debugger. From the affected samples, we discovered that only 61% have the debugger check function as its immediate root cause (i.e., they quit right after the debugger check. This result shows that once again a minority but still a significant number of samples perform additional environment checks after the debugger detection before deciding to quit execution.

Divergence due to network dependency. Samples that depend on network interactions often present the ability to connect to distinct servers, which might be uncovered by multipath execution approaches. Most network-related malware samples present two key execution points that can be severely affected by failures and variations: name resolving and connection procedures. The effects of variations on domain names are often reported in the literature via the examples of botnets and DGA algorithms [81]. In our experiments, 37% of all samples use domain name-related APIs (e.g., `gethostbyname`). However, we did not observe any significant effect on variations in the API return of these functions, which affected 0.6% of the samples. In turn, we observed a significant effect on samples related to socket openings. In our experiments, 33% of the samples use explicit connection APIs (e.g., `connect`), such that 54% of them are directly affected by changes in function returns. In 18% of the cases, IP addresses after a first failed connection attempt disappeared from the trace (in comparison to a reference one). In another 14% of the traces, new IP addresses appeared in subsequent traces after a first connection failure, thus showcasing that multipath execution approaches can reveal hidden IP addresses used as backup servers in malware samples. Interestingly, we observed that for our dataset, the strategies of checking domain name API returns and connection API returns are disjoint, with only 0.7% of all samples implementing both checks simultaneously.

Divergence due to resource unavailability. Another reason for a sample presenting diverging execution paths is its resilience against resource unavailability. When a sample is developed having defensive programming in mind, the sample will perform checks to ensure the success of a given action (e.g., API function call) and it will implement multiple distinct mechanisms to perform the same actions, such that they can be used interchangeably, thus mitigating errors caused by particular failure cases. In our experiments, we identified that a major cause for

diversions due to resource unavailability was the impossibility to access filesystem resources, which affected 19.7% of all diverging samples. On the one hand, this result corroborates previous work's hypothesis that most malware samples do not properly check for correct execution states [82]. On the other hand, this result shows the existence of many samples resilient against minor failures. Divergence cases were easily identified due to the trace size increase, which happens because of the sample's internal constructions in the form `while(ReadFile()!=SUCCESS)`, that cause a trace size explosion when the invoked function always fails. We believe this result is important because even though the majority of samples are not affected, it highlights that multipath execution can reveal details about the sample's internal implementation choices.

Divergence via environment idiosyncrasies. The root cause for the path divergence in some samples is not related to the sample's implementation but to the underlying platform they are implemented in. For instance, in Windows, there are versions of the same APIs to support ANSI (A) and Wide chars (W) argument strings, such that these distinct APIs might be invoked depending on the executed path. In our experiments, we identified this effect in 27 traces related to 6 samples. Whereas multiple samples invoked both A and W API versions during their runs depending on the data encoding to be handled, in the observed diverging cases, the function arguments handled via the W APIs were the same ASCII-encoded strings handled via A APIs. This might be related either to (i) idiosyncrasies related to the code generation process, or (ii) attackers purposely using this type of construction to deceive some kind of analysis procedure (not necessarily multipath ones). In both cases and even though this is a very small effect, we believe this report is important since as far as we know this type of divergence in executed paths has not been reported in the literature before. We believe that this observation was only possible due to the use of a larger dataset in comparison to previous studies, which sheds light on rarer events.

Divergence effect over malware families. Although each sample presents its own characteristics, according to the individual project and implementation decisions, some general conclusions can be drawn according to the families the samples belong to, as some general behaviors implemented by the families are commonly affected by divergences in API function returns. For instance, consider three reasonably popular malware families in our dataset: Autoit (1%), Injector (5%), and Downloader (27%). When we isolate the root causes of path divergence to cover only deviations due to changes in the return value of debugging functions, a prevalent root cause, we notice that the samples from the Injector family are much more affected (73% of the diverging samples in this family are affected by this cause) than the Autoit (5%) and Downloader (29%) families. We hypothesize this high incidence to be related to the nature of the Injection behavior, which is a natural target for reverse engineers, thus requiring protection. This same reasoning does not apply to Downloaders, whose real target for reverse engineers is the downloaded payload, thus requiring fewer protections. In turn, when we isolate the root cause to be deviations in the result of network-related API function calls, we observe the opposite result. The Downloader family was the most affected (50% of all diverging samples) since this type of sample is totally dependent on the network for its operation. We did not observe any variation caused by network effects on Autoit and Injector families, which can be explained by their non-reliance on network functions. Based on these results, we conclude that some families are more affected by multipath execution than others, according to the malicious behaviors implemented by the families.

5.2 Divergence effect on ML-based malware classifiers

We previously identified that the number of evasive samples in the dataset is $\approx 3\%$. Also, some more non-evasive samples presented alternative execution paths of some kind, even though error ones. It is plausible and often hypothesized that these paths affect detection, but it is unclear in the literature what is the real impact of evasive samples and alternative execution paths. Would it be proportional to the rate of intentionally evasive samples or not? To answer this question, we conducted the classical malware detection experiment described in most papers but now considering traces with multiple, different execution paths. Ideally, we would like to test the impact

based on all techniques, but not all analysis techniques produced results for all samples, which limits the training of models for them. Therefore, to perform fair experiments, we decided to limit this experiment to the fuzzing technique, which covered all samples.

We implemented for our tests a model previously described in the literature [83] to rely on its previous validation. The proposed experiment consists into training the model with traces that present no external stimulation for variation (traditional sandbox trace) and predicting the 10 first traces fuzzed at different rates. Both training and test sets included evasive and non-evasive samples. Both sets include traces for all (the same) samples. The rationale behind it is that if the fuzzed paths (prediction) are similar to the non-fuzzed ones (training), the detection rate should be maximal (100%). In turn, if the fuzzed paths actually affect the detection, some samples might not be detected despite the model knowing the original trace.

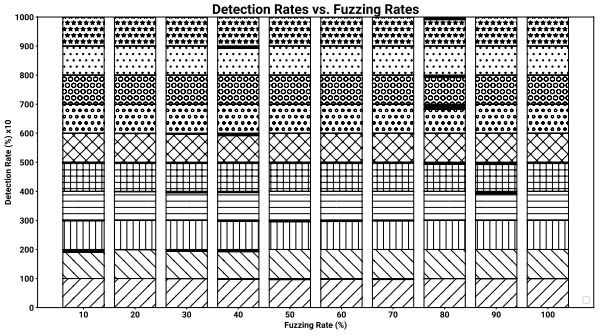

Fig. 7. **The impact of fuzzing on ML-based malware detection.** Detection (hatched) and evasion (black) rates for varying amounts of fuzzing.

Figure 7 shows the detection (hatched) and evasion (black) rates achieved in each trace. We initially observed a low number of evasions, as the first trace is still very close to the original traces. The more randomness is added to the traces, the more evasions are observed. However, as previously noticed for the trace diversity, a large amount of randomness does not increase evasion. The traces with the largest amount of fuzzing (100%) were not evasive as they present low diversity, being mostly limited to error paths. Intermediate amounts of fuzzing led to the highest evasion rates.

The overall impact of multipath execution on ML-based malware detection can be considered as **infrequent** (less than 1% in most configurations), but **non negligible**. Further analysis revealed that all the evasive samples (3%) really evaded detection. More than that, we noticed that samples previously detected also escaped detection in a few cases (up to 8% for the 80% fuzzing amount). This phenomenon indicates that the detection previously only took place in the trace regions that varied with fuzzing and not due to a global understanding of the entire execution intent as malicious. This weakness of the ML models is derived from their pure probabilistic nature, which highlights the need for creating malware classifiers more focused on invariants.

6 DISCUSSING & MOVING FORWARD

In this section, we first discuss our findings and this work's limitations. Later, we present actionable items to move the field forward.

Novelty. To the best of our knowledge, our work is the first to compare the different multipath malware tracing approaches. More than comparing, we went further and scaled the experiments from dozens or hundreds to thousands of samples, which allowed us to increase the confidence in the reported rates of samples presenting

diverging behavior. Our study is also the first to consider the results over an in-the-wild set of samples, rather than a selection of evasive samples, thus providing a more realistic view of the prevalence of evasive samples. Our work not only presents a literature review but also demonstrates with experiments how the theoretical drawbacks of each technique appear during analysis and points out their impact on the analysis practices.

Implementation Limitations. The present experiments target the general case, such that we did not implement the handling of special cases–e.g., we fuzzed all APIs with the same chance. Adaptive approaches are left as future work. Similarly, the evaluation of the performance of specific optimizations for each class of solutions is left as future work. We believe that the adopted approach of generalizing the results of one solution to its entire class is reasonable for our goal of presenting an overview of the techniques because we compare the solution's characteristics and not their individual performance. In this sense, when we claim, for instance, that a fuzzer solution produces at most one new path per run vs. symbolic executors producing multiple new paths per run, this observation is true for any fuzzer or symbolic executor implementation.

Evaluation Limitations. Though we presented extensive experiments, our analyses are not exhaustive. Further experiments must evaluate the generalization of the results for other platforms (e.g., Android/Linux), tools, and datasets–Experiments with our dataset reveal approaches' drawbacks, but do not establish formal boundaries.

Existing Development Opportunities. As with any analytical research, we do not propose new techniques. Thus, there is an emerging number of new techniques to still be evaluated, such as (1) porting analysis from the Android environment [9, 51] to desktop applications; (2) making analysis faster (e.g., Xforce [94], Tainting [58]); and (3) increasing symbolic executors robustness to packing [8].

Transition to Practice. Though the techniques presented in this work are well-known by academia, they are not easily available to end-users. Although most malware researchers rely on automated sandboxes for analysis purposes, we could not find a single public platform able to exercise multiple execution paths. Thus, developing practical, public multipath execution platforms is key to moving the field forward.

6.1 Moving Forward

We following systematize the identified gaps in actions to be addressed by all community's stakeholder to move the field forward.

- For Malware Analysts:
  - Consider running the samples multiple times and under different conditions, as a non-negligible number of samples present evasive paths and most samples present alternative execution paths in general.
  - Remember that even samples that initially presented no alternative execution paths might have hidden paths that are revealed when properly stimulated.
- For Sandboxes Providers:
  - Integrate multipath execution capabilities in the solutions, as they are not available in the existing commercial products despite their popularity in the academic literature.
- For Detection Solution Providers:
  - Train and test detection models with multiple traces for the same samples, as the execution diversity affects the results in practice.
- For Researchers:
  - Analysis Solutions' Developments:
    - Develop path exploration strategies specific for malware analysis, as this task requires meeting different properties from vulnerability discovery.
    - Add support for child processes' tracking in symbolic executors to avoid missing a significant number of deviation cases.

    * Develop in-memory state forking for parallel path exploration in concrete executors without the need for instantiating multiple VMs.
    * Develop adaptive fuzzing strategies to benefit from the different impacts that different APIs cause in multipath execution.
    * Develop ML-based detection mechanisms more focused on finding global maliciousness invariants rather than focusing on local, statistical patterns.
- Evaluation Developments:
    * Develop clear criteria for malicious paths to allow the comparison of tools on the same grounds.
    * Consider only the code section of the binaries in the multipath evaluations, as the libraries' internals introduce their own sources of variations.
    * Develop unified metrics that allow to integrate the information about code coverage and new paths, as they might not be aligned.
    * Develop strategies to measure coverage in the presence of dead code and packed code, as these effects bias the results by artificially increasing and decreasing the absolute values.
    * Identify the ideal amount of fuzzing to maximize the discovery of new paths.

7 CONCLUSION

We investigated the problem of multipath execution for malware analysis. We presented a critical analysis of published academic works to highlight aspects that must be improved toward making the task more practical (e.g., making snapshot reversion faster to scale analyses, creating new metrics for evaluating code coverage in the presence of dead code, and the need for better path prioritization strategies for malware symbolic execution). We corroborate these findings with results from experiments with real malware samples. We expect to help foster the development of next-generation analysis solutions.

Reproducibility. The code developed for this research project is available at https://github.com/marcusbotacin/MalwareFuzz

Acknowledgments. Marcus Botacin thanks NSF for the support via the CNS 2327427 grant.

REFERENCES

[1] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J. . Lalande, and V. Viet Triem Tong. 2015. GroddDroid: a gorilla for triggering malicious behaviors. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, US, 119–127. https://doi.org/10.1109/MALWARE.2015.7413692

[2] Vitor Afonso, Anatoli Kalysch, Tilo Müller, Daniela Oliveira, André Grégio, and Paulo Lício de Geus. 2018. Lumus: Dynamically Uncovering Evasive Android Applications. In *Information Security* (2018-01-01), Liqun Chen, Mark Manulis, and Steve Schneider (Eds.). Springer International Publishing, Cham, 47–66. https://secret.inf.ufpr.br/papers/lumus.pdf

[3] Mohammed Noraden Alsaleh, Jinpeng Wei, Ehab Al-Shaer, and Mohiuddin Ahmed. 2019. *gExtractor: Automated Extraction of Malware Deception Parameters for Autonomous Cyber Deception.* Springer, US, 1.

[4] Erin Avllazagaj, Ziyun Zhu, Leyla Bilge, Davide Balzarotti, and Tudor Dumitras. 2021. When Malware Changed Its Mind: An Empirical Study of Variable Program Behaviors in the Real World. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, US, 3487–3504. https://www.usenix.org/conference/usenixsecurity21/presentation/avllazagaj

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Assisting Malware Analysis with Symbolic Execution: A Case Study. In *Cyber Security Cryptography and Machine Learning*, Shlomi Dolev and Sachin Lodha (Eds.). Springer, US, 1.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (may 2018), 39 pages. https://doi.org/10.1145/3182657

[7] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2010. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. IFIP, San Diego, CA, 1.

[8] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, US, 633–651. https://doi.org/10.1109/SP.2017.36

[9] Luciano Bello and Marco Pistoia. 2018. Ares: Triggering Payload of Evasive Android Malware. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, US, 2–12.

[10] L. Bergroth, H. Hakonen, and T. Raita. 2000. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*. Springer, US, 39–48. https://doi.org/10.1109/SPIRE.2000.878178

[11] The Fuzzing Book. 2021. Fuzzing APIs. https://www.fuzzingbook.org/html/APIFuzzer.html.

[12] Marcus Botacin, Hojjat Aghakhani, Stefano Ortolani, Christopher Kruegel, Giovanni Vigna, Daniela Oliveira, Paulo Lício De Geus, and André Grégio. 2021. One Size Does Not Fit All: A Longitudinal Analysis of Brazilian Financial Malware. *ACM Trans. Priv. Secur.* 24, 2, Article 11 (jan 2021), 31 pages. https://doi.org/10.1145/3429741

[13] Marcus Botacin, Marco Zanata Alves, Daniela Oliveira, and André Grégio. 2022. HEAVEN: A Hardware-Enhanced AntiVirus ENgine to accelerate real-time, signature-based malware detection. *Expert Systems with Applications* 201 (2022), 117083. https://doi.org/10.1016/j.eswa.2022.117083

[14] Marcus Botacin, Fabricio Ceschin, Ruimin Sun, Daniela Oliveira, and André Grégio. 2021. Challenges and pitfalls in malware research. *Computers & Security* 106 (2021), 102287. https://doi.org/10.1016/j.cose.2021.102287

[15] Marcus Botacin, Lucas Galante, Paulo de Geus, and André Grégio. 2019. RevEngE is a Dish Served Cold: Debug-Oriented Malware Decompilation and Reassembly. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium* (Vienna, Austria) *(ROOTS'19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/3375894.3375895

[16] Marcus Botacin and André Grégio. 2022. Why We Need a Theory of Maliciousness: Hardware Performance Counters in Security. In *Information Security*, Willy Susilo, Xiaofeng Chen, Fuchun Guo, Yudi Zhang, and Rolly Intan (Eds.). Springer International Publishing, Cham, 381–389.

[17] Marcus Botacin and André Grégio. 2021. Malware MultiVerse: From Automatic Logic Bomb Identification to Automatic Patching and Tracing. https://doi.org/10.48550/ARXIV.2109.06127

[18] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. *Automatically Identifying Trigger-based Behavior in Malware*. Springer, US, 1.

[19] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 463–469.

[20] Marcel Böhme, Laszlo Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. .

[21] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babi ć, and Dawn Song. 2010. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, US, 413–425.

[22] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[23] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. 2007. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Secure Systems Lab at Stony Brook University, Tech. Rep* 1, 1 (2007), 1–18.

[24] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. 2008. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Diego Zamboni (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 143–163.

[25] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, USA, 380–394. https://doi.org/10.1109/SP.2012.31

[26] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, US, 711–725. https://doi.org/10.1109/SP.2018.00046

[27] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[28] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-Box Concolic Testing on Binary Code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. ACM, US, 736–747. https://doi.org/10.1109/ICSE.2019.00082

[29] Yehonatan Cohen and Danny Hendler. 2018. Scalable Detection of Server-Side Polymorphic Malware. *Knowledge-Based Systems* 156 (2018), 113–128. https://doi.org/10.1016/j.knosys.2018.05.024

[30] Jedidiah R. Crandall, Gary Wassermann, Daniela A. S. de Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. 2006. Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1168857.1168862

[31] Lajos Cseppento and Zoltan Micskei. 2015. Evaluating Symbolic Execution-Based Test Tools. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, US, 1–10. https://doi.org/10.1109/ICST.2015.7102587

[32] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro. 2020. On the Dissection of Evasive Malware. *IEEE Transactions on Information Forensics and Security* 1, 1 (2020), 1.
[33] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, US, 1. https://www.usenix.org/conference/woot20/presentation/fioraldi
[34] GIAC. 2004. Diffusing a Logic Bomb. https://www.giac.org/paper/gsec/3504/diffusing-logic-bomb/105715.
[35] GNU. 1991. GNU DiffUtils. https://www.gnu.org/software/diffutils/.
[36] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. *Queue* 10, 1 (jan 2012), 20–27. https://doi.org/10.1145/2090147.2094081
[37] Google. 2019. american fuzzy lop. https://github.com/google/AFL.
[38] David Goransson. 2016. Escaping the Fuzz: Evaluating Fuzzing Techniques and Fooling them with Anti-Fuzzing. https://publications.lib.chalmers.se/records/fulltext/238600/238600.pdf.
[39] André Ricardo Abed Grégio, Vitor Monte Afonso, Dario Simões Fernandes Filho, Paulo Lício de Geus, and Mario Jino. 2015. Toward a Taxonomy of Malware Behaviors. *Comput. J.* 58, 10 (2015), 2758–2777. https://doi.org/10.1093/comjnl/bxv047
[40] Adrian herrera. 2018. Analysing "Trigger-based" Malware with S2E. https://adrianherrera.github.io/post/malware-s2e/.
[41] jjyg. 2020. Metasm. https://github.com/jjyg/metasm.
[42] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. 2011. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *2011 IEEE Symposium on Security and Privacy*. IEEE, US, 347–362. https://doi.org/10.1109/SP.2011.41
[43] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-Fuzzing Techniques. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1913–1930. https://www.usenix.org/conference/usenixsecurity19/presentation/jung
[44] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. 2009. Emulating Emulation-Resistant Malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security* (Chicago, Illinois, USA) *(VMSec '09)*. Association for Computing Machinery, New York, NY, USA, 11–22. https://doi.org/10.1145/1655148.1655151
[45] Raghudeep Kannavara, Christopher J Havlicek, Bo Chen, Mark R Tuttle, Kai Cong, Sandip Ray, and Fei Xie. 2015. Challenges and opportunities with concolic testing. In *2015 National Aerospace and Electronics Conference (NAECON)*. IEEE, US, 374–378. https://doi.org/10.1109/NAECON.2015.7443099
[46] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (London, England, UK) *(VEE '12)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2151024.2151042
[47] Dhilung Kirat and Giovanni Vigna. 2015. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 769–780. https://doi.org/10.1145/2810103.2813642
[48] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 287–301. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat
[49] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking Internet Malware. In *2012 IEEE Symposium on Security and Privacy*. IEEE, US, 443–457. https://doi.org/10.1109/SP.2012.48
[50] Qiang Li, Yunan Zhang, Liya Su, Yang Wu, Xinjian Ma, and Zeming Yang. 2018. An Improved Method to Unveil Malware's Hidden Behavior. In *Information Security and Cryptology*. Springer, US.
[51] Zimin Lin, Rui Wang, Xiaoqi Jia, Jingyu Yang, Daojuan Zhang, and Chuankun Wu. 2016. ForceDROID: Extracting Hidden Information in Android Apps by Forced Execution Technique. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, US, 386–393. https://doi.org/10.1109/TrustCom.2016.0088
[52] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. 2011. Detecting Environment-Sensitive Malware. In *Recent Advances in Intrusion Detection*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 338–357.
[53] Serena Lucca. 2022. Let's analyze malware with sy.
[54] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034
[55] Haoyu Ma, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin Gao, and Chunfu Jia. 2015. Control Flow Obfuscation Using Neural Network to Fight Concolic Testing. In *SecureComm*, Vol. 152. Springer, US, 287–304. https://doi.org/10.1007/978-3-319-23829-6_21
[56] Meteorix. 2020. PyLCS. https://pypi.org/project/pylcs/.
[57] Miasm. 2020. Miasm. https://miasm.re/.

[58] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. 2016. StraightTaint: Decoupled offline symbolic taint analysis. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, US, 308–319.

[59] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 757–768. https://doi.org/10.1145/2810103.2813617

[60] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE, US, 231–245. https://doi.org/10.1109/SP.2007.17

[61] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. ACN, US, 421–430. https://doi.org/10.1109/ACSAC.2007.21

[62] Dorottya Papp, Levente Buttyán, and Zhendong Ma. 2017. Towards Semi-Automated Detection of Trigger-Based Behavior for Software Security Assurance. In *ARES (ARES '17)*. ACM, US.

[63] Kyuhong Park, Burak Sahin, Yongheng Chen, Jisheng Zhao, Evan Downing, Hong Hu, and Wenke Lee. 2021. Identifying Behavior Dispatchers for Malware Analysis. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 759–773. https://doi.org/10.1145/3433210.3457894

[64] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 829–844. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng

[65] Sebastian Poeplau and Aurélien Francillon. 2019. Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and Its Generation. In *ACSAC (ACSAC '19)*. ACM, US.

[66] Google ProjectZero. 2018. WinAFL. https://github.com/googleprojectzero/winafl.

[67] Xiao Qu and Brian Robinson. 2011. A Case Study of Concolic Testing Tools and their Limitations. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, US, 117–126. https://doi.org/10.1109/ESEM.2011.20

[68] Quarkslab. 2020. Triton. https://triton.quarkslab.com/.

[69] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*. IFIP, US, 1.

[70] Christian Rossow, Christian Dietrich, and Herbert Bos. 2013. Large-Scale Analysis of Malware Downloaders. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Ulrich Flegel, Evangelos Markatos, and William Robertson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 42–61.

[71] J. Samhi and A. Bartel. 2022. On The (In)Effectiveness of Static Logic Bomb Detector for Android Apps. *IEEE Transactions on Dependable and Secure Computing* 1, 01 (aug 2022), 1–1. https://doi.org/10.1109/TDSC.2021.3108057

[72] Jordan Samhi, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2021. Difuzer: Uncovering Suspicious Hidden Sensitive Operations in Android Apps. arXiv:2112.10470 [cs.CR]

[73] Cuckoo Sandbox. 2015. Hooked APIs and Categories. https://github.com/cuckoosandbox/cuckoo/wiki/Hooked-APIs-and-Categories.

[74] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*. SSTIC, Rennes, France, 31–54.

[75] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2021. Loki: Hardening Code Obfuscation Against Automated Attacks. arXiv:2106.08913 [cs.CR]

[76] Stefano Sebastio, Eduard Baranov, Fabrizio Biondi, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. 2020. Optimizing symbolic execution for malware behavior classification. *Computers & Security* 93 (2020), 101775. https://doi.org/10.1016/j.cose.2020.101775

[77] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P*. IEEE, US, 1.

[78] Maksim Shudrak. 2018. Fuzzing Malware For Fun and Profit. https://www.youtube.com/watch?v=bzc44WPxoxE.

[79] Chengyu Song, Paul Royal, and Wenke Lee. 2012. Impeding Automated Malware Analysis with Environment-sensitive Malware. In *7th USENIX Workshop on Hot Topics in Security (HotSec 12)*. USENIX Association, Bellevue, WA, 1. https://www.usenix.org/conference/hotsec12/workshop-program/presentation/Song

[80] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. IFIP, US, 1–16.

[81] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. 2009. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. Association for Computing Machinery, New York, NY, USA, 635–647. https://doi.org/10.1145/1653662.1653738

[82] R. Sun, M. Botacin, N. Sapountzis, X. Yuan, M. Bishop, D. E. Porter, X. Li, A. Gregio, and D. Oliveira. 2020. A Praise for Defensive Programming: LeveragingUncertainty for Effective Malware Mitigation. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2020), 1–1. https://doi.org/10.1109/TDSC.2020.2986112

[83] Ruimin Sun, Xiaoyong Yuan, Pan He, Qile Zhu, Aokun Chen, André Grégio, Daniela Oliveira, and Xiaolin Li. 2022. Learning Fast and Slow: Propedeutica for Real-Time Malware Detection. *IEEE Transactions on Neural Networks and Learning Systems* 33, 6 (2022), 2518–2529. https://doi.org/10.1109/TNNLS.2021.3121248

[84] TheRegister. 2017. IT plonker stuffed 'destructive' logic bomb into US Army servers in contract revenge attack. https://www.theregister.com/2017/09/22/it_contractor_logic_bombed_army_payroll/.

[85] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. 2019. A Close Look at a Daily Dataset of Malware Samples. *ACM Trans. Priv. Secur.* 22, 1, Article 6 (jan 2019), 30 pages. https://doi.org/10.1145/3291061

[86] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking Flaws in Systems Security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, US, 310–325. https://doi.org/10.1109/EuroSP.2019.00031

[87] X. Wang, Y. Yang, and S. Zhu. 2019. Automated Hybrid Analysis of Android Malware through Augmenting Fuzzing with Forced Execution. *IEEE Trans. Mob. Comp.* 1, 1 (2019), 1.

[88] Wired. 2015. Logic Bomb Set Off South Korea Cyberattack. https://www.wired.com/2013/03/logic-bomb-south-korea-attack/.

[89] Michelle Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *NDSS (NDSS)*. IFIP, US. https://doi.org/10.14722/ndss.2016.23118

[90] H. Xu, Z. Zhao, Y. Zhou, and M. R. Lyu. 2018. Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs. *IEEE TDSC* 1, 1 (2018).

[91] Zhaoyan Xu, Lingfeng Chen, Guofei Gu, and Christopher Kruegel. 2012. PeerPress: Utilizing Enemies' P2P Strength against Them. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 581–592. https://doi.org/10.1145/2382196.2382257

[92] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. 2014. GoldenEye: Efficiently and Effectively Unveiling Malware's Targeted Environment. In *Research in Attacks, Intrusions and Defenses*, Angelos Stavrou, Herbert Bos, and Georgios Portokalidis (Eds.). Springer International Publishing, Cham, 22–45.

[93] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *CCS (CCS '15)*. ACM, US.

[94] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, US, 1121–1138. https://doi.org/10.1109/SP40000.2020.00035

[95] Chaoshun Zuo and Zhiqiang Lin. 2017. SMARTGEN: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 867–876. https://doi.org/10.1145/3038912.3052609