PFUZZER: Practical, Sound, and Effective Multi-path Analysis of Environment-sensitive Malware with Coverage-guided Fuzzing

Nicola Bottura, Daniele Cono D'Elia, Leonardo Querzoni
Sapienza University of Rome
{bottura, delia, querzoni}@diag.uniroma1.it

***Abstract*—Among the behaviors and tactics that malware can exhibit, environment-sensitive logic likely poses the longest-standing challenge to the analysis capabilities of automatic systems such as sandboxes. Current analysis approaches either fall short in anticipating adversarial tactics by design, or incur prohibitive costs and other roadblocks when reasoning on real-world code. As a result, manual analysis remains the primary way to identify behaviors that show only when a machine meets specific expectations of the sample.**

To address these issues, we present the first practical, sound, and effective solution for multi-path exploration of environment-sensitive malware. We argue how the popular coverage-guided fuzzing paradigm from software testing can effectively achieve this task, provided we can devise original design solutions (such as coverage feedback and environment mutations) tailored to the unique characteristics of malware to enable this application. Our approach not only can disarm many evasions without requiring expert knowledge, but also unveil additional activities that would not show in a baseline run due to environmental conditions unrelated to evasion.

We build a manually annotated dataset of environment-sensitive malware and use it to estimate the analysis capabilities of the approach. Our PFUZZER implementation reveals activity that the best competitor misses for 36.09% of the samples: such activity either follows evasions that deceive existing systems or comes from behaviors that show only in other "right" environments. PFUZZER also unveils dormant evasive tactics for 70.64% of the samples that one may wrongly deem as non-evasive after a baseline run.

***Index Terms*—Malware, evasion, fuzzing, anti-analysis.**

1. Introduction

Malware historically represents a critical threat in the cybersecurity landscape. To examine the behavior of an untrusted executable, being static analysis alone often inconclusive, modern defense approaches employ dynamic analysis for execution inspection and tracing. Many solutions assume that a single execution is sufficient for characterization [1], but this easily becomes a slippery road in the presence of environment-sensitive malware.

Environment-sensitive malware can show relevant behavioral differences depending on the characteristics of the machine where it runs. With the so-called *evasive* malware, nowadays increasingly prevalent [2], [3], samples may elude analysis through, for example, early termination if they recognize distinctive features of analysis environments. However, behavioral differences may also originate from operational characteristics of the machine, either for abusing them (for example, a specific process for data-stealing malware) or as a distinctive feature of intended victims for targeted malware. While researchers have been tackling these problems for well over a decade [4], this type of malware still strongly challenges current defenses.

We can divide current approaches in two kinds: approaches that build increasingly transparent environments by removing artifacts known to tip off evasive malware, and approaches that study how the instructions from the sample interact with information from the environment and try to force alternative decision outcomes. All these approaches may be limited in the type of fingerprinting techniques and decision logics they can handle, require manual extensions to face new techniques, or face scalability barriers that hinder their use on real-world malware.

We find all these issues to root into two methodological pitfalls: relying on predetermined expectations for a sample's actions and attempting fine-grained analysis of code. The former is problematic because an outcome for an action may not fit all samples and not all actions are foreseeable; the latter because code analysis is expensive and adversaries can make it arbitrarily more challenging.

At a closer look, these challenges share similarities with what software testing literature has lately done, very successfully, with coverage-guided fuzzing for testing programs. While these programs are not adversarial, fuzzers typically do not know how an input looks like or what makes an input good for a particular program. However, thanks to input mutations and run-time feedback of general applicability, they progressively learn through a plurality of executions what a program can accept.

Readers knowledgeable in fuzzer design would argue that, in spite of the conceptual similarities, the leading techniques in fuzzing (from input acquisition and mutations to the coverage feedback or the seed scheduling strategies) may not be applicable to malware or likely yield low efficacy. This paper bridges this gap by proposing and evaluating techniques to unleash coverage-guided fuzzing for analyzing environment-sensitive malware.

We first model alternative environments as an input construction process, introducing *execution policies* as a unifying umbrella to interpose and alter the course of a sample's accesses to environmental information sources, enacting *mutation* rules that are the core of the policy. Then, to track progress in building meaningful environments for the sample, we define a *coverage feedback* that is sensitive to externally observable effects from execution (as these characterize malware activity the most), but also inspects code coverage to not miss any partial progress. External coverage separately accounts for accesses to

environmental sources (as these accesses are potential candidates for mutation) and conspicuous actions that alter the machine state (as the presence of new actions may suggest that some expectations were met). As we will detail in the paper, coverage facts are also essential for us to drive mutations and to orchestrate fuzzing scheduling.

Unlike existing multi-path proposals based on forced execution [5], our runs are always sound: analysts or downstream systems can apply the prescriptions from a policy and reproduce all the behaviors independently. Unlike suggested approaches based on symbolic execution or taint analysis, we refrain from analyzing code, but only observe it in a (limited) number of executions. Unlike most existing solutions, we do not need expert knowledge with predetermined answers or to make assumptions on tactics, but we try to automatically synthesize the "right" outcomes for queries using only a coverage feedback. This gives us a solution that is practical, sound, and—as the experimental evaluation will show—very effective, too.

To estimate the capabilities of our approach, which we implement in a system called PFUZZER[1], we build a manually annotated dataset of 1078 samples of potentially independent interest for researchers. We pick these sample through deduplication, clustering, and reversing work starting from 315,231 Windows malware samples collected by qualified sources between 2018 and 2023.

We study how PFUZZER can unveil additional behaviors for these samples and compare its performance with two state-of-the-art systems, BLUEPILL [6] and ENVIRAL [7], that build on expert knowledge. PFUZZER emerges as the clear winner, for example unveiling conspicuous activity that the best competitor misses for 36.09% of the samples. This activity may originate from evasive tactics that PFUZZER neutralizes, but more often is additional activity that adds to the conspicuous actions from the baseline run. Analyses based on a single run, which are the de-facto standard [1], would likely miss the latter behaviors. Furthermore, our method is also able to trigger dormant evasive activity in samples that already reveal all their malicious activity in the baseline run. Our experimental analysis will discuss several statistics and trends we identify for the alternative paths we unlock, and evaluate internal performance drivers of our approach.

In summary, this paper proposes as contributions:

- A practical and sound method for analyzing environment-sensitive malware with fuzzing;
- New designs for key fuzzer components (mutation, coverage, scheduling) tailored to malware;
- A dataset of environment-sensitive samples with annotations on the tactics we spotted in them;
- An experimental analysis that estimates the capabilities and performance of our PFUZZER system and provides insights on the alternative behaviors it unveils for these samples and on key internal aspects behind PFUZZER's performance.

The dataset and the experimental materials from this paper are available at https://github.com/Sap4Sec/pfuzzer/.

2. Background and Motivation

Environment-sensitive malware has been studied for long in the security literature. Pioneering seminal works like [4] characterize it in terms of sandbox evasion tactics: that is, malware that is able to recognize when running in an analysis system rather than on a real user's machine and thus refuses to perform its intended malicious course. In this early literature, researchers acknowledge how it is fundamentally impossible to build a transparent analysis system [8]; hence, defensive research has to be mostly reactive: once sandbox developers become aware of new evasion tactics, they tweak their system to thwart them [4].

These researchers focus their efforts also on accurately detecting environment-sensitive malware by looking for meaningful discrepancies in execution behaviors among runs attempted in multiple analysis systems [4] or using an uninstrumented reference host [9], [10].

But even if perfect detection methods were available, for many malware authors the goal is not to prevent manual analysis of samples, but rather prevent analysts from using popular efficient automated systems [11]. Therefore, researchers continue to build increasingly transparent methods that, albeit imperfect, reduce the "attack surface" for evasive tactics. Solutions of this kind include moving most of the analysis logic to privileged opaque layers (through Virtual Machine Introspection [12]–[14] or using System Management Mode [15]) and supplying fake indicators in response to know evasions tactics [6]. Neither are generally reliable: for example, virtualization artifacts [16] and timing side channels [17], [18] give away VMI-based solutions, whereas fake indicators are effective until adversaries come up with zero-day evasions [6].

An important aspect to underline is that environment-sensitive malware is not necessarily evasive. While evasive malware is important and increasingly prevalent [2], [3], many samples choose not to show their behaviors depending on characteristics of the execution environments unrelated to artifacts from analysis systems. Notably, *targeted malware* [19] is designed to fly under the radar and enact its intended behavior only on machines that exhibit distinctive characteristics known to be in place at the intended victim's (e.g., a company or government entity). The solutions we discussed above for evasive malware are, by design, incapable of handling such malware.

More interestingly, as we analyze throughout our experimental evaluation, much malware that already shows conspicuous behaviors in a baseline run (hence, one would not consider it evasive or targeted) can reveal additional, and at times distinctly new, behaviors when one or more features of the surrounding environment change. We can think, for example, of OS version, timezone and language information, hardware devices, running and installed applications, recently opened documents, and so on.

Multi-path exploration may thus appear as natural way to handle general environment-sensitive malware. To date, a few approaches have been discussed or explored for it.

One is *forced execution* [5], [20], which flips every branch instruction and massages process memory for best-effort repair of runtime errors (as random data typically ends up into instruction operands along the coerced paths). This approach shows valuable results for tasks like control-flow graph recovery [5] and malware unpacking [21]. However, it suffers from severe limitations when handling general code, such as state explo-

1. The name hints at how we fuzz what a malware sample perceives.

sion [21], sticking to fixed paths within OS libraries [5], and unsoundness [5] from exploring unfeasible paths. To date, the technique has been demonstrated only on Linux malware or ad-hoc case studies (e.g., SPECINT 2000).

Another approach is to use fine-grained program analyses, such as symbolic execution and taint analysis, to identify dependencies in malware code from environmental information items a sample retrieves at run-time. The pioneering work of Moser et al. [22] labels inputs of interest and tracks their propagation throughout execution, taking snapshots at control-flow decisions dependant on a labeled value; then, in a new execution, it uses a linear constraint solver to try to generate a value that would flip the branch. The method is more sound than forced execution for the feasibility of the explored paths, but is neither fully sound nor complete. It then faces high run-time costs and potential imprecision from the taint tracking, and general limitations with constraint solving. Similar limitations, and even higher costs, affect potential approaches hinted around the use of symbolic execution, which recent work deems as impractical [1]. Additionally, anti-analysis constructs known for both program analyses [23], [24] can further marginalize multi-path approaches that rely on accurate modeling of code and execution semantics.

In summary, the state of the art struggles in two respects. Solutions that conduct "single-path" analysis are helpful (yet still limited, and needing manual tweaks over time) against evasive malware, but cannot predict the right answers for other types of environment sensitivity. Solutions for multi-path analysis could potentially handle general environment-sensitive malware, but the pitfalls and costs for accurate modeling of code are substantial.

To tackle this conundrum, we look into cross-fertilization opportunities with a methodology that has significantly advanced the state of the art in software testing in recent years, addressing challenges that, on the surface, share similarities with those above. Through a plurality of executions under a lightweight instrumentation, *coverage-guided fuzzing* solutions learn how to build increasingly complex inputs that reach different program parts without requiring expert knowledge or accurate modeling of code. Obviously, the domains and nature of programs under analysis are very different, and as we will see malware analysis calls for original fuzzer design techniques for such an approach to be effective. This paper will propose and evaluate contributions in this direction, showing how to build a practical, sound, and effective fuzzing solution for multi-path analysis of environment-sensitive malware.

3. Design

To tackle the challenges from Section 2, we design and implement a coverage-guided fuzzer, PFUZZER, around the specificities of environment-sensitive malware.

Overview. We propose a design where a fuzzer gradually mutates how the sample perceives the surrounding environment and recognizes meaningful differences in the sample's execution through a coverage mechanism. As typically with coverage-guided fuzzing, the system maintains a queue of alternative environments that led to previously unseen execution states and chooses between them to generate the next environments, hoping to progressively meet more and more of the sample's expectations.

For such a fuzzer, the concept of input to mutate becomes: *What aspects of the environment should we alter as the sample interacts with them?* Hence, we interpose on accesses to sources of environmental information (as with OS APIs and certain CPU instructions), and we decide whether to apply mutations to each of them to alter their course: for example, to simulate the absence or presence of a file. We term these inputs *execution policies* and instantiate each policy as a set of mutation rules.

During execution, a runtime monitor realizes the interposition to enact any mutation in the policy, but also to collect a coverage feedback for the execution. The goal is to detect whether the policy is, in fuzzing terminology, *interesting* (that is, it reveals novel coverage) and thus retain it. For each policy, we compute a significance score to estimate the progress and an energy score to control how often we may attempt to evolve the policy further.

A scheduler component uses the two scores to balance exploration across different policies. This avoids that the fuzzer is led astray by partial progress in a direction that will eventually be inconclusive (that is, the fuzzer can backtrack), or that it may miss alternative malicious behaviors for different environments (for example, with alternative tactics in the presence of particular system configuration, running programs that may be abused, etc.).

In the following, we discuss how to build an effective coverage feedback and soundly mutate environments, then we move to fuzzer design and implementation choices and discuss other methodological aspects of our approach.

3.1. Coverage Feedback

The first significant challenge to face is determining to what extent the execution environment presented to a sample meets the characteristics and run-time behaviors the sample expects. This challenge is shared with virtually all automatic approaches to malware analysis and is a hard one to tackle because no ground truth is available.

Generally speaking, malware fingerprints an environment by accessing one or more *items of environmental information* and then running at a later point a decision code on their contents. What complicates the analysis is that such code can be of arbitrary complexity.

In some cases, malware may act upon the retrieved information only long after a query. Additionally, malware may employ obfuscation and other adversarial techniques (e.g., implicit flows [23]) to impede code analysis. Malware can also access multiple sources at once and build an opaque summary of the results before making a decision, acting upon the value of the summary without providing obvious clues on what items do not meet its expectations. Furthermore, malware can access some items without acting upon their contents, for the purpose of deflecting the focus of an analysis system or human operator.

All these techniques (and variants not discussed here for brevity) further marginalize all current techniques available for analyzing environment-sensitive malware.

In our work, we take a radically different approach. By repeatedly massaging (as detailed next) the environment a sample perceives, we obtain a plurality of executions that help us estimate how we are progressing in building environments that meet the sample's expectations. Our goal is to collect execution facts that are meaningful also in the

presence of adversarial tactics like those described above. Therefore, we radically move away from prior endeavors that study code fragments for their potential outcomes and how to enforce them (e.g., [1], [22]) or that enact other low-level manipulations in a similar fashion [19].

To estimate exploration progress, we define a *coverage feedback* mechanism that tracks execution facts observable at run-time with modest overhead. In the adversarial setting we target, conventional feedback sources that are effective in other domains can lose efficacy or effectiveness. For example, if we study only code coverage, a malware writer can embed code that conditionally activates stalling sequences or other deceptive behaviors, and uncovering these regions may give a false sense of progress.

To comprehensively capture the execution of a malware sample, we track facts stemming from two sources:

- *external coverage*: information about externally observable actions on the machine by the sample, focusing on (i) environmental queries and (ii) changes made to the state of the machine;
- *internal coverage*: information on the internal program states traversed by the execution.

External Coverage. As we mentioned, our feedback source draws from two types of externally observable behaviors. For the first, we track what environmental information items the sample accesses through queries. This includes the invocation of operating system APIs for querying the environment (e.g., about files, processes, registry contents, hardware interfaces, timing sources, GUI windows, and system settings) and the use of certaint CPU instructions that provide direct access to environmental information (e.g., `cpuid` and `rdtsc`).

For the second, we track both transient and permanent changes brought about by sample execution to the global state of the machine. Transient changes include, for example, attempted network communications and the creation of temporary IPC objects such as mutexes. Permanent changes include creation, modification, or removal of, e.g., files, processes, GUI windows, and system settings.

Internal Coverage. This feedback source includes information about internal program states, such as code portions traversed by the sample during one execution: for example, the identity of the basic blocks covered. Optionally, internal coverage may include some data flows, such as input arguments for code whose semantics is to compare strings or other byte sequences.

Discussion. We argue both coverage sources are helpful and that not considering either may reduce the efficacy of our method, due to the risk of overlooking behaviors.

If we ignore external coverage, internal coverage alone may not suffice to suggest us if we progressed (that is, if we met some of the sample's expectations about the environment). Newly observed queries may be a consequence of how we massaged the environment, supplying the right values to the decision logic that otherwise impedes these queries. Changes brought about to machine state are also a natural, and possibly stronger, indicator of progress.

Unfortunately, there are cases where newly made queries are a decoy, or where there are no new visible effects on the machine because the environment does not meet yet some condition in some decision points of the sample. Here, internal coverage can greatly aid and inform the exploration when new parts of the sample are gradually met (and even suggest values for massaging the environment differently in later attempts). However, internal coverage is not strong enough to do away with external coverage, as it can be misled (for example, when reaching code present in a sample only for misleading an analysis) and can be inefficient (as it is cannot discriminate the presence of notable interactions with the environment).

3.2. Mutating the Environment

The second significant design challenge to face is how we can alter what a samples perceives of the environment in a productive way for our purpose. The coverage information described in the previous section will allow us to effectively distinguish whether a sample behaves differently when faced with two (possibly related) different environments. However, to be able to present a sample with such environments, we introduce two key elements in our design: the concept of *execution policy* and the *execution monitor* component that we use to enforce the execution policy and also collect coverage information.

Execution Policy. An execution policy (or policy for brevity) is a set of rules that instructs the execution monitor how to alter, before their retrieval, the contents of environmental information items that the sample accesses. The goal is to provide *feasible* outcomes for these queries and have the execution monitor capture through the coverage feedback how the sample reacts to it.

A rule prescribes how to alter the normal operation of specific accesses, for example by modifying the contents that the OS produces in response to a query, by denying the access, or by forging a fake result for an operation that the system would normally deny. A rule is not concerned with the identification of what program instructions may act on the contents of the retrieved information.

Being a policy a set of rules, it can be empty. In this case, the operation of the machine is unaltered and the execution monitor only collects coverage information during execution. Such information eventually becomes integral to the policy: in particular, we annotate each policy with external coverage information on the accesses the sample makes to environmental information when executing under that policy. Due to this choice, we are able to derive a new policy from the current one by applying one or more fuzzing-style *mutations* to the current set of rules.

Mutations. A mutation takes as input an execution policy and the external coverage information under such policy. Operationally, for each access recorded as coverage information, we check whether the original policy contains one or more rules involving the access and act accordingly.

When no such rule exists, we may nondeterministically choose to add a mutation realizing one of the options above: that is, modifying the contents of the involved environmental information item, denying an access that the machine would normally allow, or forging a fake result to simulate a successful access. However, we may also choose not to add a mutation, leaving the access unaltered.

When one or more such rules exist, each of them can be, if applicable, altered in its value-related contents, replaced with another rule, removed, or left unaltered.

Mutations are designed to be aware of the structure of the environmental information. They can add, modify, or delete portions of the contents by means of operators (e.g., bit and byte flipping, substitution with random or controlled contents) that preserve the layout and type of the involved environmental information item. Mutations for primitive values can apply to one or more constituent bytes, whereas for structured values—such as with a list of files or processes—to their structure, too (by, e.g., adding, duplicating, splitting, merging, or deleting list items).

Execution Monitor. The execution monitor component fulfills two tasks: recording coverage information and manipulating the operation of accesses to environmental information items if prescribed by the execution policy.

Both tasks require interposition capabilities realizable through standard means in dynamic program analysis.

When recording external coverage for accesses to environmental information, the execution monitor collects not only the operation (instruction or API) behind the access, but where it originates in the sample, what input arguments the operation receives, and the exit status, if applicable. We can use this information to generate rules that, in subsequent executions, can capture occurrences of said access and manipulate its operation as desired.

When altering the operation of an access, we ensure that what the sample sees is a feasible outcome for a properly configured machine. For example, if the sample tries to access a file that does not exist, we do not simply change the status code of the operation, but we rewire the access to a synthetic file that we generate on the spot. This provision mitigates the risk of spurious executions.

Discussion. An inherent focus of our design is to identify (and, as we discuss next, further explore) states and behaviors of the sample under analysis that have not been seen before. To this end, we use coverage information and mutation rules to derive new execution policies that, in turn, enable executions of the sample while changing its perception of the surrounding environment at each step.

At the same time, through coverage information, we can identify execution policies that lead to previously unseen internal states or externally observable actions for the sample. Our method operates in a fully automated manner to (i) test the end-to-end effects, on the sample under analysis, of the values it retrieves for the environmental information items it accesses, and to (ii) generate accordingly execution policies that can expose new behaviors. As we discuss throughout the illustration of our design, we can achieve step (ii) by progressively learning from past executions done under different environment policies.

The attentive reader may observe that different rules may prescribe different behaviors when the sample accesses the same information through distinct operations (e.g., using different APIs). This is a deliberate design feature that enables us to extract insights on a sample's implementation for potential integrity checks on the environment. Through subsequent mutations, our approach can achieve *eventual consistency* by having a policy where coherent rules regulate all distinct accesses to the item.

Optionally, mutations may benefit from internal coverage information on recorded data flows. For example, recording strings or byte sequences that the sample supplies to comparison functions may provide clues on elements that a mutation can hide or materialize in values produced by the system, analogously to the input-to-state correspondence analysis of [25] for overcoming fuzzing roadblocks. However, we found no benefits from enabling this strategy with the samples we tested (and therefore we disable it by default), as the process we described above for generating execution policies could apparently already handle their queries. Nevertheless, this strategy may be helpful with some targeted malware or other complex cases that expect specific values in structured outputs.

3.3. An Efficient Fuzzer for Environments

The elements presented in the previous two sections enable us to bridge most of the existing gap for applying the concept of fuzzing to the analysis of untrusted software with potential environment-sensitive behavior. In the following, we detail a possible design for a fuzzer that, leveraging both elements, can effectively discriminate and in turn efficiently explore a plurality of execution paths to unveil environment-sensitive behavior from malware. Figure 1 succinctly depicts the operation of PFUZZER.

An Atypical Fuzzer. Existing fuzzer designs cannot directly take on malware samples for analysis for a number of reasons. As we discussed, in an adversarial setting, conventional feedback schemes such as code coverage can lose efficacy. However, a good sensitivity of the feedback scheme is essential for effectively deploying modern fuzzers. We thus presented a mechanism that can effectively capture progress (and drive policy mutations) as we recognize and eventually counter multiple fingerprinting attempts. As we discuss next, this helps us mitigate the risks that the fuzzer is led astray by adversarial patterns.

Next, while programs that normally undergo fuzz testing have well-defined input sources, for malware we proposed the concept of execution policy as a unifying umbrella to interpose on program accesses to multiple variable sources of environmental information and alter their normal course. By modifying the environment the sample perceives, we may force it to take different, environment-sensitive execution paths without having to modifying or analyzing its code. The collected coverage information on the issued queries guides, in an adaptive and automatic fashion, the selection of what parts of the environment we may try to alter for the next runs.

While the two elements above can enable us to build a fuzzer that can effectively elicit and differentiate execution behaviors, there are a few design choices to make in order to ensure that the fuzzer is also *efficient*. These choices will be the focus of the remainder of this section.

Epochs and Policy Generation. Classic coverage-guided fuzzers derive test cases from one another through mutations: while they expose the derivation history (for example, through filename conventions in AFL-style fuzzers), they typically allow mutations to do unrestricted changes to the test case structure. This choice may be inefficient in our scenario, as we may lose track of, and inadvertently undo, mutations that contributed new coverage.

Our design evolves execution policies, which are our equivalent of test cases, in distinct *epochs*. We try to strike a balance between exploration (i.e., reaching and recognizing new behaviors) and exploitation (i.e., keep mutating fruitful policies hoping to unveil further behaviors).

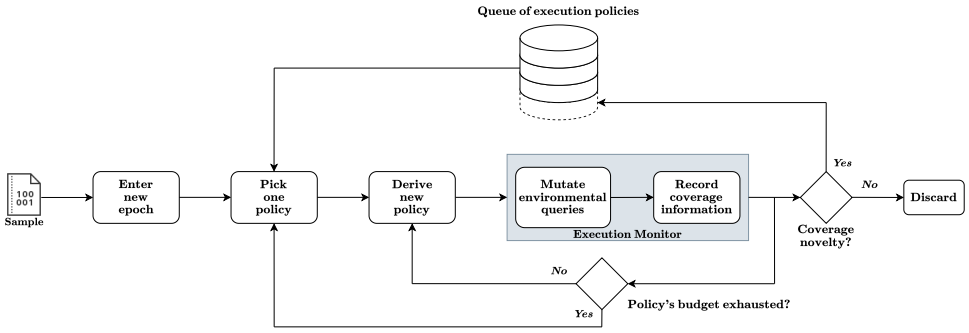


Figure 1. Workflow of PFUZZER: deriving new execution policies and checking the coverage feedback for retaining interesting policies in the queue.

At each epoch, we select one policy from the *queue* of policies the fuzzer maintains and derive new policies from it. Each epoch has a budget of executions to attempt and a time limit. A new policy inherits the set of rules from its parent. Then, we extend the set by choosing nondeterministically, for each access to environmental information recorded in the policy's coverage annotations, whether to apply one mutation from those applicable to the access.

For each generated policy, we run the sample and collect coverage information, looking for novelty compared to the plurality of past executions, using a *coverage map* to distinguish such novelty. If a policy brings new coverage, we add the policy to the queue. Once the budget runs out, the fuzzer enters a new epoch and the process restarts.

Scheduling. The fuzzing process starts with an empty execution policy or optionally with one supplied by a human operator (e.g., the result of a previous analysis). For all the epochs that follow, the scheduling algorithm for selecting a policy from the queue plays a key role for efficiency. For each policy we retain in the queue, we use a *significance score* to prioritize the selection and an *energy score* for defining the budget for the epoch if selected.

As we discussed, each epoch can generate one or more policies due to novel coverage. The significance score of such a policy is the weighted sum of external coverage for changes to the system, external coverage for environmental information accesses, and internal coverage:

$$w_1 * actions_{sys\_change} + w_2 * actions_{query} + w_3 * BBs$$

counting unique $actions$ of a kind and covered basic blocks BBs. Weights decrease in value from w_1 to w_3: the rationale is to favor, as a proxy for progress, activity that is externally observable and more conspicuous.

Then, at the end of an epoch, we sort any newly added policies by significance score, then further increase this score for the top ranked ones (top-2 in the implementation) and decrease it otherwise. We assign energy scores following the same order, with higher budget for the top ranked policies and equal budget for the others, if any. The rationale is to favor policies that bring other coverage compared to their parent, but limiting the risk of biasing future exploration to prioritize all such related children.

The energy score of a policy defines the possible budget for an epoch if selected. At the end of the epoch, we reduce the policy's energy score to account for the exploitation (i.e., mutation) work attempted on the policy at hand. Therefore, we may eventually be left with policies with high significance score but depleted energy scores, which will lead the fuzzer to pick other policies.

When no policy has energy left, we process the queue and assign policies with a predefined energy score, starting with those from older epochs. This approach facilitates backtracking by giving the fuzzer another chance to revisit previously fuzzed policies, hoping to uncover new execution paths through possibly different choices of mutations.

3.4. Implementation

Our design requires interposition capabilities for accesses to environmental information and tracing capabilities for internal coverage. For practicality, we implement PFUZZER using dynamic binary instrumentation (DBI). We choose the DynamoRIO [26] framework over Intel Pin [27] as, in our experience, Pin has higher startup latency with just-in-time compilation of a sample's code.

To focus exclusively on a sample's actions, we use an interval tree as in [28] to ignore API calls and low-level instructions issued within OS library internals. We then use call-site information to discriminate multiple invocations of an action from different places in the sample.

Coverage. For interposing on accesses to environmental information, we target a selection of 68 APIs (counting `A/E/ExA/ExW` variants as one) that prior work [6], [7] deems as relevant. When applicable, we track relevant arguments (e.g., output buffers) to alter when fuzzing. These APIs cover multiple environmental information sources, including but not limited to files, processes, registry, drivers, hardware strings, time sources, and networking.

We also instrument `cpuid` and `rdtsc` instructions, as they expose device and temporal information that malware notoriously uses for fingerprinting [6]. We ignore other instructions, such as `int` and `popfd`, popular in anti-debugger detections, as those would pertain to possibly imperfect instrumentation from the DBI engine[2].

For tracking transient or permanent changes to the machine state, we monitor 146 APIs that we obtain from a popular source [30] by applying slight refinements.

For internal coverage, similarly to standard fuzzers, we maintain a map for each execution and one that cumulatively accounts for all the basic blocks traversed in previous runs. Upon execution completion, we compare the two maps to recognize if the policy unveiled at least one new basic block compared to all other policies. For

2. Such gaps are more rare in DynamoRIO than in Pin [29].

simplicity, we use these maps also to detect when new APIs falling under our categories of interest are invoked.

Mutations. To regulate the course of an environmental information access, we implement four general operators:

- *Normal course:* the access operates normally;
- *Force success:* if an access would normally fail (e.g., a file does not exist), make provisions so it can succeed (e.g., rewire it to a dummy file);
- *Force failure:* the converse of the previous case;
- *Retarget:* replace a value pertaining to the access with a different, meaningful one (i.e., a value that induces feasible execution outcomes[3]). Depending on the nature of the access, this value can be an input parameter or an output element (e.g., a name appearing in an output list of running processes).

We apply mutation operators by considering only those sound for a given operation (that is, the OS may allow this outcome on a different machine). For example, if a sample queries the username with the `GetUserNameA` API, we can only allow its normal course or retarget its output to show a different username string.

For time-related operations, we use dedicated operators to alter the course of delay primitives (zeroing or rescaling the quantities) and the elapsed time from query operations (to artificially materialize the expected delays). Unlike [6] and most sandboxes, we do not follow a fixed strategy, but at each run we nondeterministically choose (and record) if to apply adjustments and in what amounts.

Execution. To orchestrate executions, we use one driver process that creates (and destroys upon timeout) a new process for each run of the sample. This choice, although not ideal for fuzzing throughput[4], preserves execution soundness. When a samples has derived execution flows, such as child processes, we interpose on their accesses to environmental information and use the same mutation rules encoded in the policy for the initial process.

3.5. Discussion

Our approach aims to enable practical multi-path execution of malware through the coverage-guided fuzzing paradigm. To deal with environment-sensitive behaviors, we propose a feedback mechanism sensitive to different sources of progress (new environmental queries, new conspicuous actions, new code executed) and use it to conduct a biased exploration of the space of countless alternative environments. This process unveils "faulty" environments where the sample hides its true colors, as well as "right" environments where its intended activities unfold, possibly with significant differences between right environments.

Significance scores allow us to prioritize execution policies by analyzing properties of their coverage novelty, while energy scores let us balance the exploration among multiple meaningful alternative scenarios if present. This design makes our approach valid not only for evasive malware (for which typically just one "right" path exists), but also for samples exhibiting a variety of behaviors influenced by environment-related factors. As several researchers find this diversity to be commonplace in much malware [1], [31], [32] but current "single-path" systems miss it, our approach may also benefit downstream tasks like classification that we do not consider in this study.

As our experimental analysis will show, PFUZZER shows to be practical and effective, outclassing other solutions against evasive malware and revealing additional behaviors for samples that already show malicious activity in a baseline run. In this paper, we do not claim to have solved the malware evasion problem: as we discuss next, PFUZZER comes with implementation and design aspects that adversaries may try to target. Our goal is to show that coverage-guided fuzzing can be a solid and profitable new approach to multi-path analysis, avoiding the major road blockers of existing designs proposed for it (Section 2). After our paper, other works may follow and evolve its core concepts with additions and improved techniques.

Limitations. We acknowledge the following limitations.

In a practical setting, every dynamic analysis has a time budget. Adversaries aware of our design may deploy stalling sequences not based on delay primitives. The problem is general, but such procrastination may be detected [34] and a better implementation could use live snapshots instead of starting all runs from the entrypoint.

Whereas the fuzzing entropy takes care of selecting what APIs to mutate and with what values, we rely on (limited) expert knowledge for compiling a list of APIs and their relevant arguments for mutation. An adversary may target an API not in our list. While adding new APIs is simple, this can be automated in two ways. One extension may look up prototypes and try to blindly mutate new APIs observed at runtime. A smarter version may leverage large language models to parse the MSDN documentation and identify what invoked APIs are related to the environment and synthesize sound inputs for them.

Researchers explored anti-fuzzing techniques to impede security auditing [35], [36]. With existing methods, our design either counters them (as with SpeedBump [35], BranchTrap [35], and fake code [36]), or is unaffected due to differences in feedback and expected outcomes (i.e., crashes). Nevertheless, adversaries may eventually devise techniques tailored to specific implementation choices, or uncover a general assumption that PFUZZER depends on; we failed to identify obvious cases at the time of writing.

Finally, our method may struggle with malware that expects very specific values in outputs that are not general (e.g., a particular username) and with malware that relies on indirection (meaning such values should appear within, for example, files that it opens). For the former, Section 3.2 discusses how internal coverage for dataflow facts may expose these values to our mutations through input-to-state correspondence [25]. For the latter, we believe our approach could be extended to model indirection and expose these contents to the fuzzer's inner workings.

As for implementation limitations, DBI is notoriously imperfect for transparency [37], although many use it to analyze evasive malware (e.g., [2], [3], [6], [38]). We did not attempt to patch conspicuous traits of Dy-

3. When the set of possible outputs is limited (e.g., keyboard layouts), PFUZZER draws replacement values from (partial) lists of values embedded in its implementation. For collections (e.g., list of processes) or resources, it selects from predetermined entities (e.g., fixed strings, paths to typed decoy files). For simpler cases (e.g., GetCursorPos), it chooses values randomly within ranges permitted by the argument semantics.

4. Fuzzing mechanisms like forkservers or snapshots [33] would help but embodiments for Windows user code are currently missing or far less mature than on Linux. We are confident this will change in future.

namoRIO [29], also because the samples we tested stress them much more rarely (if at all) than what expected with Pin's defects [6]. However, with adequate engineering effort, PFUZZER can be ported more transparent schemes, for example atop modified hypervisors [39]. Our goal was showing that the fuzzing angle is feasible and profitable.

Similarly, adversaries may try to hide API calls using exotic techniques [40] or target corner cases that our mutations miss (e.g., with new timing attacks). We expect implementation additions would mitigate these threats.

4. Dataset Construction

Challenges. A notable challenge in malware research is the difficulty to establish a ground truth for analysis tasks on real-world samples. This is particularly true for environment-sensitive malware, for which notable works that study the evasion phenomenon [2], [38] or propose systems to combat it [6] typically draw conclusions based on indicators collected by an analysis system for one execution of each sample. A pitfall of this approach is that, without knowing what techniques each sample embeds, evaluations can accurately estimate only the relative performance of multiple systems, and not the general effectiveness of a specific method under scrutiny.

With manual reverse engineering currently representing the only accurate means to recover such information, building a sizeable annotated dataset for evaluations at scale remains an open challenge for security researchers.

To mitigate this gap and foster future research in the area, we built a curated dataset of 1078 PE32 samples with environment-sensitive behavior. Each sample underwent manual scrutiny and comes with the list of behaviors (evasive and post-evasion) from our multi-path analysis or, if its results were inconclusive, with our findings from manual reverse engineering of what impeded its analysis.

Our dataset size means to strike a balance between the expensive manual reverse engineering process and the significance of the collection. The size is comparable to what state-of-the-art work [6] on combating evasions uses with manual result review, but comes with a greater heterogeneity (e.g., we avoid skewing effects from near duplicates, present instead in [6] as recently noted in [7]).

Collection. Our dataset includes samples collected in the wild between 2018 and 2023 from two main sources. We start from the VirusTotal Academic dataset, which offers samples first spotted in the wild between 2018 and 2021 and flagged as malicious by at least 15 anti-virus products in use to VirusTotal. For more recent years, we turned to the Bazaar collection from VX Underground, a dataset updated on a monthly basis with samples collected and reported as malicious by industry professionals.

We start from 315,231 samples and apply a number of filtering steps to identify and retain environment-sensitive samples to form a collection as rich in heterogeneity as possible while also tractable in size for manual inspection.

The **first** filtering step is to identify possible near-duplicates using structural analysis. We rely on the *vhash* value, an in-house similarity clustering algorithm value that VirusTotal uses for structural comparisons. After this step, we are left with 71,151 samples (including 2,908 for which VirusTotal has not computed a vhash value yet).

The **second** filtering step is to retain samples with conspicuous traits that hint at environment-sensitive behavior. This step is necessary as not all malware is environment-sensitive: we chose to prioritize, as in prior studies [6], samples that show these traits to pattern-based static analysis[5]. We start from public Yara rules [41] for detecting anti-analysis behavior and enact the following actions:

1) By testing all rules, we are left with 40,189 samples (56.48%) matching at least one rule;
2) We remove samples matching only rules for anti-debugging behavior, as they have limited appeal for our purpose. Evasions exclusive to debuggers (e.g., mishandling of single-step exceptions) target implementation gaps in the monitoring technology: the correct behavior is the same for any malware analysis system, and enforcing it upon divergence is a one-time implementation effort. This choice leaves us with 12,341 samples.
3) We adjust the rules to remove patterns for specific sandboxes or hardware fingerprints of hypervisors other than VirtualBox as they would not affect our test environment (Section 5). We are now left with 15 rules for queries about drivers, files, firmware strings, hardware features, libraries, and processes, and a total of 2,018 samples.

The **third** filtering step involves looking statically [42] for samples (163) written in managed languages such as .NET and removing them, as done in other studies [2]. This is because analyzing managed languages rather than native code poses challenges orthogonal to our purpose[6]. At this point, we are left with 1,855 samples.

The **fourth** filtering step involves balancing over-represented malware families. We cluster samples into families using AVClass [43], [44] and limit to 10 the number of samples per family in each year. Additionally, we limit the number of occurrences within families (and in the remaining unlabeled samples) of executable shielded by the same executable protector, using also version information to differentiate protectors when available. We use the fingerprints of [42] to identify such protectors. At this point, we are left with 1,262 samples from 239 families.

The **fifth** and final filtering step involves dynamic execution of the samples to identify cases overlooked by the static analysis tools we used. In particular, we spot another 16 .NET samples. Additionally, we identify 46 samples that run 64-bit compiled code, albeit they are marked 32-bit in their original dataset. Due to their spurious nature in otherwise heterogeneous, large collections of 32-bit malware, we remove these as well. The resulting dataset contains 1200 likely environment-sensitive samples.

During the last step, we also identify 122 samples that hit two peculiar implementation defects of DynamoRIO. While we reported these DBI cache coherency bugs to its maintainers, they have not been solved to date. We found

5. This choice can lead us to exclude samples with environment-sensitive activity missed by the patterns. However, the alternative would be to retain and examine manually both code and run-time activities of many more samples, which would be an untenable effort for our purpose.

6. For our design, we would have to add provisions for tracking internal coverage within CIL bytecode in the runtime and filtering out API calls not originating from CIL bytecode. Nevertheless, in preliminary experiments with our (unmodified) system, we tested .NET samples with results consistently better than with the systems we test in this paper.

that all these samples are shielded with the Themida protector using a specific option selection. For our evaluation, we preferred to leave out these sample from the dataset. However, we refer interested readers to Appendix D, where we show how a partial porting of our approach to Pin already handles most of those samples successfully.

Overall, our final dataset features 1078 samples, with 138 unlabeled and 940 attributed to one of 239 families.

5. Evaluation

Using the samples from our dataset, we estimate the capabilities of PFUZZER by conducting a set of experiments to answer the following research questions:

- **RQ1:** Can our approach expose environment-sensitive behavior in real-world malware?
- **RQ2:** What is the performance of PFUZZER compared to state-of-the-art solutions?
- **RQ3:** How can PFUZZER build environments with the very characteristics samples are sensitive to?
- **RQ4:** What alternative behaviors can we observe for a sample when analyzed in PFUZZER?

Analysis Environment. To conduct the experiments, we follow state-of-the-art practices (e.g., [6], [45]) to assemble virtual environments with characteristics and wear-and-tear state as realistic as possible, including but not limited to applications, documents, and usage history.

We use Windows 10 64-bit version 1803 with English as system language. To remove potential hurdles for some samples, we disable the User Account Control (UAC), enable the Administrator account, and run each sample with elevated privileges. To prevent potential harm, we disconnect the machine from the internet and deploy, as commonly in the field, a local subnet with a network simulator on a REMnux VM using INetSim and PolarProxy.

We run our experiments on a physical server equipped with an Intel Xeon CPU E5-2620 v3 @ 2.40GHz CPU, 48 GB of RAM, Linux OpenNebula with kernel 5.4.0. We deploy 5 sibling VMs built as above, assigning each with 2 CPU cores and 4 GB of RAM. We start the analysis of each sample from a common live VM snapshot, taken with negligible background activity ongoing in the VM. We use VirtualBox 7.0.10r158379 to host these VMs.

Conspicuous Activity. To estimate the behavior detection capabilities of the systems we test, we build an automated metric to recognize conspicuous activity that we can attribute with confidence to malicious behavior. With this metric, we seek to automate a semi-qualitative assessment of recorded activity, mimicking the work of professionals when writing reports and avoiding quantitative assessments based on uninformed API counts as in [7] as they can be severely misleading (Appendix A). Due to space limitations, we refer to Appendix B and to the evaluation materials for a richer explanation of the method. In short, we mine execution logs for a subset of the 146 APIs (Section 3.4) behind system changes for external coverage, prioritizing APIs we can most strongly associate with malware activity. We complement this with an analysis of patterns indicative of specific malware behaviors (e.g., injection, persistence), taking into account API sequences, parameters, and frequency of invocation. The output is list of relevant conspicuous activities in a given execution.

5.1. RQ1: Unveiling Additional Behaviors

In the following, we study how PFUZZER can expose environment-sensitive behaviors of a sample compared to a baseline run in the same machine. For each sample, we allocate a budget of 10 minutes for the analysis, allowing up to 55 seconds (Section 5.3 discusses this choice) per single execution and leaving up to 5 seconds for log analysis and snapshot restoration. We choose 10 minutes to target the upper limit for how malware analysts use automatic sandboxes. With this budget, we are guaranteed to perform at least 10 different executions, including the baseline run. In our experiments, though, we typically observe many more (we measure them in Section 5.3) due to earlier termination along evasive paths.

Table 1 reports the breakdown of the results for the dataset according to the *additional* behaviors we identify when fuzzing. We qualitatively group the samples by the *conspicuous* activity they achieve across all runs in the mutated environments. As conspicuous, we consider what we obtain from our evaluation metric that qualitatively analyzes transient and permanent effects on the system from malware execution. We identify five groups:

- **No extra activity.** Baseline run shows conspicuous activity, fuzzing does not expose more of it.
- **Mild additions.** Baseline run like above, fuzzing exposes new conspicuous activity but we cannot pin it to general behaviors not surfaced before.
- **Strong additions.** Baseline run like above, fuzzing exposes conspicuous activity that clearly speaks of behaviors (e.g., injection, keylogging, persistence, network connections) missing in the baseline run.
- **Highly evasive.** The baseline shows no conspicuous activity, fuzzing exposes conspicuous activity that we can pin to typical malware behaviors.
- **Inconclusive.** The baseline run shows no conspicuous activity, fuzzing may expose alternative paths but none shows significant conspicuous activity.

The baseline run is the initial 55-second execution in PFUZZER with an empty execution policy. When we identify new activity, we repeat and extend the baseline run to 8 minutes to validate that we did not miss such activity because of a too short baseline execution.

Samples with *No extra activity* account for 31.91% of the dataset. However, our method is valuable for their analysis: as we show in Section 5.4, while our machine does not trigger their evasive tactics, PFUZZER shows for 70.64% of them alternative paths that lead to evasion (e.g., when an analysis tool is running): knowing about their existence and what triggers them is a valuable finding.

Samples with *Mild additions* account for 13.17% of the dataset (142 samples). As notable new behaviors, we mainly observe file (113 samples) and registry manipulations (37), and occasional system service changes (3). These are unlikely to show in single-path analysis systems.

Samples with *Strong additions* account for 15.68% of the dataset. We find them particularly interesting, as they often unveil behaviors that are completely missing in the baseline run. Section 5.4 will analyze their nature in detail.

Samples in the *Highly evasive* set show little to none activity in a baseline run and account for 13.54% of the dataset. We consider them highly evasive as, besides a few

	Category	Samples
	No extra activity	344 (31.91%)
457 (42.39%)	*Mild additions*	142 (13.17%)
	Strong additions	169 (15.68%)
	Fully evasive	146 (13.54%)
Inconclusive 277 (25.69%)	Cmd-line arguments	11 (1.02%)
	DynamoRIO bugs	19 (1.76%)
	External dependencies	51 (4.73%)
	PoC evaders	5 (0.46%)
	Interactive	49 (4.54%)
	Unable to start program	67 (6.21%)
	Undetermined evasions	75 (6.95%)
	Total	1078

TABLE 1. BREAKDOWN OF SAMPLES BY ADDITIONAL BEHAVIORS EMERGED VIA FUZZING AND CONFIRMED BY MANUAL ANALYSIS.

file manipulations (Section 5.4), they do not cause changes to system state. They may tip off a classic analysis system for the environmental queries they make, but the analysis (besides dubbing them suspicious) would be inconclusive.

These numbers suggest us that, for malware authors, environment-sensitive tactics are not necessarily (and often are not) a one-time decision before showing conspicuous behaviors. If existing "single-path" systems were to analyze the samples we classify as with mild or strong additions, they would likely report them as malicious to users, but with an incomplete output that leaves the identification of these additional behaviors to manual work.

Overall, our method reveals additional conspicuous behaviors for 42.39% of the dataset (457 samples). No action is apparently required for 31.91% of it (344), but the analyses for RQ4 will prove this hypothesis wrong.

We term *Inconclusive* the remaining samples (277) as we were unable to see conspicuous behaviors for them. These account for 25.69% of the dataset. We conducted a manual analysis to determine which of these samples we fail to analyze due to limitations of PFUZZER.

We identify 19 samples triggering DynamoRIO bugs and 75 samples for which we suspect evasion activity but neither our analysis nor reports from sandboxes in our availability revealed sufficient details on the employed techniques. Overall, limitations of PFUZZER affect only 94 of the tested samples at most (8.72% of the dataset).

Most of the remaining 183 samples (17.07% of the dataset) require user interaction (e.g., filling text forms or using installers), specific command-line arguments for the executable, or external dependencies (configuration files, DLLs typical of videogames, specific network traffic); the rest may fail to start (corrupted files or run-time Windows errors) or be innocuous PoC evaders sent to VirusTotal.

The analysis of these samples requires provisions orthogonal to our goals and to comparable solutions. To the best of our knowledge, for some challenges (e.g., external dependencies with files and DLLs) the malware analysis community has yet to propose methods to tackle them.

However, all these samples are representative of malware in the wild. We include them in this experiment to avoid over-claiming the efficacy of PFUZZER, and annotate them in our dataset to hopefully ease endeavors from researchers working on the challenges behind them. Appendix C provides further context for these samples.

To conclude the analysis of the research question, we studied the per-family variability of the capabilities of PFUZZER discussed above. We found that, for the samples we keep in the dataset for them, the majority of families (69.04%) fall within a single category. When a family has samples in multiple sets, the most recurrent sets are those with mild and strong additions: at a first look, this suggests that such families tend to show environment-sensitive behavior that is not strongly related to evasion, but rather comes from behavioral variability [1], [32].

5.2. RQ2: Comparison against State-of-the-art

In this section, we compare the performance of PFUZZER against two systems: BLUEPILL and ENVIRAL.

BLUEPILL [6] is a state-of-the-art solution for building increasingly transparent analysis environments via fake answers based on expert knowledge. In case studies with highly evasive malware, its authors claim performance on par, and at times better, with commercial sandboxes.

ENVIRAL [7] lies between the approaches of PFUZZER and BLUEPILL. While its authors present it as a fuzzer, the system speculatively evolves a single path in repeated executions using predetermined answers against known evasions and random outputs for other accesses. ENVIRAL retains a random output choice if it leads to a higher end-to-end API call count, without allowing backtracking. Its authors claim performance superior to BLUEPILL.

In our study, PFUZZER largely outperforms BLUEPILL and ENVIRAL. Additionally, ENVIRAL is the least performant system, in apparent contradiction with the preliminary evaluation in its paper. As we explain in Appendix A, its study has three issues: a clerical error when modifying BLUEPILL, an unsound budget choice for BLUEPILL to offset DBI overheads, and a coarse-grained evaluation metric also prone to distortion effects. BLUEPILL's paper comes instead with no progress or activity metric.

We use our evaluation metric and compare the two systems with PFUZZER, focusing on the run that reveals the highest amount of conspicuous activity. For each sample, we inspect what environmental queries PFUZZER manipulates in the run of interest: if the other system does not have provisions for capturing such a query, we run an ablated version of PFUZZER where we disable the manipulation. This ablation step is needed only for ENVIRAL with `cpuid` and `rdtsc` instructions, whereas the capabilities of the three systems are functionally equivalent for intercepting all the API-based queries we recorded.

As both systems natively target Windows 7, we build for them a Windows 7 SP1 32-bit VM with characteristics as the Windows 10 one we used before. Then, for every sample on which PFUZZER outperforms either, we repeat our analysis on the Windows 7 VM to confirm or reassess the case (i.e., removing rare OS-dependent discrepancies).

In the comparisons, we do not cover the 183 samples (Section 5.1) featuring orthogonal challenges that none of the systems can overcome. However, we validated no relevant differences exist in their analysis reports for them.

BluePill. BLUEPILL does a single-execution analysis. As it uses Intel Pin, which in our experience has higher initial latency than DynamoRIO, we cautiously allow it to execute each sample for 4 minutes, which is about four times longer than one run in PFUZZER. Then, we analyze if PFUZZER finds a different amount of conspicuous activity for the sample than BLUEPILL with our metric.

Category	Better	Equal	Worse
No extra activity	28	315	1
Mild additions	66	76	0
Strong additions	135	34	0
Fully evasive	94	51	1
Undetermined evasions	0	70	5
DynamoRIO bugs	0	12	7
Total	323	558	14

TABLE 2. ACTIVITY FOUND: PFUZZER VS. BLUEPILL.

	PFUZZER			*Ablated* PFUZZER			
Category	**Better**	**Equal**	**Worse**	**B.**	**E.**	**W.**	**Σ**
No extra activity	45	285	14	46	288	17	+7
Mild additions	108	34	0	103	34	0	-5
Strong additions	150	19	0	137	19	0	-13
Fully evasive	90	54	2	68	54	2	-22
Undet. evasions	0	72	3	19	80	9	+33
DynamoRIO bugs	0	16	3	0	16	3	-
Total	393	480	22	373	491	31	-

TABLE 3. ACTIVITY FOUND: PFUZZER VS. ENVIRAL.

Table 2 summarizes the outcome of the comparison. Overall, on 895 samples, PFUZZER finds more activity than BLUEPILL for 323 samples (36.09% of the total) and less activity only for 14 samples (1.57%).

To reflect what value PFUZZER can bring for the analysis, the table further characterizes the results by the labels we gave to samples when discussing RQ1. The vast majority of the cases where the two systems are evenly matched are from the *No extra activity* group, for which PFUZZER did not need to counter any tactics. We note here how the predetermined fake answers from BLUEPILL introduce a regression 28 such samples: that is, forcibly altering the normal course of execution at each suspicious action is detrimental for their analysis[7].

PFUZZER largely outperforms BLUEPILL on samples from the *Strong additions* (better on 135 out of 169) and *Fully evasive* (better on 94 out of 146) sets. We observe no particular correlation between lower performance and the year when those samples were first spotted, so we tend to exclude obsolescence issues for BLUEPILL. We believe that the superior performance of PFUZZER stems from two issues that it side-steps by design: (i) even if from accurate expert knowledge, predetermined fake answers may not suit all samples, and (ii) aiming to cover all seemingly adversarial queries with such answers may backfire.

BLUEPILL shows more activity than PFUZZER only for 5 *Undetermined evasions* samples, for 7 of the 19 samples that we could not analyze due to occasional DynamoRIO bugs, and for 2 samples from all the other groups. For the first 5 samples, we suspect DBI-related evasions that either BluePill neutralizes through the mitigations of [37] for Pin or affect only DynamoRIO [29].

Enviral. For testing ENVIRAL, we allow a total 10-minute budget as we did for the PFUZZER tests. We set a 25-second duration per run, with a 10x increase on the original evaluation settings [7] coherently with the insights from Appendix A, which shows 2.5 seconds are a poor choice to characterize behaviors. While its authors granted BLUEPILL 4x longer runs to offset DBI overheads, for the same imbalance we give PFUZZER only 2.2x more time (55 seconds) per run than ENVIRAL (incidentally, this may potentially let ENVIRAL attempt many more executions).

The left side of Table 3 summarizes the outcome of the comparison. Overall, on 895 samples, PFUZZER finds more activity than ENVIRAL for 393 samples (43.91% of the total, for a 7.82% net difference than with BLUEPILL) and less activity for 22 samples (2.46%). As with BLUEPILL, the table characterizes the improvements with respect to the RQ1 labels we gave to samples.

ENVIRAL introduces important regressions for the samples that we categorize as *No extra activity* for PFUZZER: for 45 of them (13.08% of the group), ENVIRAL sees less activity than the baseline run of PFUZZER. This suggests that ENVIRAL interferes with execution and likely triggers evasive paths. We suspect a mix of analysis overheads, conspicuous traits of its trampoline-based hooks, and implementation bugs is behind the regression.

On the bright side, ENVIRAL unveils more activity for 22 samples in the group: by inspecting the nature of these activities in the logs, they fall under the definition that we gave in RQ1 for the *Mild additions* set, this time built around the results of ENVIRAL. Our explanation is that ENVIRAL could attempt many more executions than PFUZZER (due to higher run duration and lower instrumentation overhead) and thus unveil these paths, which however PFUZZER may still reach with more time.

PFUZZER largely outperforms ENVIRAL in exposing additional activities for the *Mild* (76.06% of its samples) and *Strong* (88.76%) groups. We believe this comes from a design limitation: ENVIRAL continues to evolve the most promising path and, as explained in its paper, it misses backtracking capabilities to explore alternative paths. Our approach relies on energy and significance scores (Section 3.3) to shift exploration between multiple directions, and benefits also from a more sensitive feedback that combines three sources to pick up indicators of progress.

For the *Highly evasive* samples, we believe the expert answers ENVIRAL relies on for bypassing common evasions allow it to match PFUZZER only for those samples (36.99%) for which the available predefined fake answers suffice, resulting in a performance similar to BLUEPILL.

To write off an implementation limitation of ENVIRAL, we conduct an additional experiment by ablating in PFUZZER the instrumentation of assembly instructions (`cpuid` and `rdtsc`) precluded to ENVIRAL's Detours-based approach. The right part of Table 3 summarizes the new comparison. To this end, we recompute the labeled sets as the capabilities of PFUZZER are now reduced; therefore, the number and identity of samples in each set can change[8]. The performance trends remain similarly unfavorable for ENVIRAL, which makes us conclude that this implementation limitation is not the main cause for ENVIRAL being the least performant system in our study. While its mixing of expert knowledge and speculative single-path exploration is interesting, the issues in its evaluation (Appendix A) and the outcome of the experiments above suggest that ENVIRAL may still need enhancements.

Sandboxes. Some of our readers may wonder why did not consider sandboxes in this study. The 146 *Fully evasive*

7. We noticed a few tricky anti-patterns, for example with BLUEPILL shortening necessary delay operations (e.g., for thread completion).

8. The *Undetermined evasions* set now sees further 33 samples for which the ablated PFUZZER misses conspicuous activity: 22 were priorly *Fully evasive*, 5 with *Strong additions*, and 6 with *Mild additions*. Other changes are from 8 samples formerly with strong additions: they become mild for 1, and none for the remaining 7 (now in the *No additions* set).

		Mild additions	*Strong additions*	*Fully evasive*
Total trials in 10' budget		43.5	47.2	103.7
Policies bringing novelty		18.85	18.73	29.06
Trials until first novelty		6.18	4.06	4.6
Trials until best policy		16.23	17.51	29.22
Mutations in best policy	Normal course	4.09	3.26	2.56
	Force success	2.26	1.78	0.95
	Force failure	2.01	1.76	1.00
	Retarget	5.75	5	3.73
	Alter time	0.61	0.50	0.42

TABLE 4. INTERNAL FUZZER METRICS FOR PFUZZER (AVG).

samples from RQ1 may suit such a comparison, as their malicious activity shows only if a sandbox counters their evasions. However, for the rest of the dataset, sandbox designs generally lack the means to automatically tweak the environment for aspects unrelated to evasion, which were instead key to revealing additional activity in samples that show maliciousness already in baseline runs.

For the sake of completeness, we still attempted an analysis of the *Fully evasive* samples in a state-of-the-art open source sandbox with anti-evasion capabilities: CAPE. We left out commercial products due to their opaque working, as it would challenge result explainability, and license agreements that prohibit benchmarking.

CAPE classifies 67.58% of these samples as malware, 20.68% as suspicious binaries, and the remaining 11.72% as benign software. Unlike our approach, CAPE applies also static signatures and other techniques (e.g., in-memory analysis) to categorize samples; nevertheless, it misjudges approximately one every three *Fully evasive* samples. Further inspection of the recorded API calls often reveals fewer behaviors than both PFUZZER and BLUEPILL do, suggesting its real performance is lower than what the classification results initially suggested.

5.3. RQ3: Behind PFUZZER's Performance

In this section, we aim to shed light on how PFUZZER can build execution policies that meet the characteristics that the samples in our study are sensitive to. We collect various metrics on the working of PFUZZER on the 457 samples (Table 1) for which it reveals new behaviors.

Time Sensitivity. Under our per-sample budget of 10 minutes, we analyze what is the impact of the maximum duration we allow per run on the efficacy of the method.

We variate the duration by 25 seconds, leading respectively to shorter (30 seconds) or longer (80 seconds) executions. The default choice of 55 seconds (and another 5 for post-processing) ensures we can conduct at least 10 runs: with the shorter duration we can now complete at least 17 runs, and at least 7 with the longer duration.

As with RQ2, we use our metric for conspicuous activity to compare the relative performance of PFUZZER. With the shorter duration, PFUZZER finds the same activity as in the default setting for 298 samples (65.21%) and less activity for 158 (34.57%). This shows that shorter runs are typically insufficient to capture behavior despite PFUZZER can do many more of them in the given budget. The additional runs unveil more activity for only 1 sample, initially in the *Mild additions* group and now in the *Strong additions* one as PFUZZER could continue to evolve a promising execution policy in the additional runs.

With the longer duration, PFUZZER finds the same activity as in the default setting for 315 samples (68.93%) and less activity for 141 (30.85%), which suggests that longer runs are unnecessary for our dataset. The exception is 1 sample for which we record new networking activity. Recent studies [46] show how it is difficult to set the right threshold for having a "single-path" sandbox collect sufficient information to classify or study a sample, but hints that most behaviors are likely observable in the first 2 minutes. We find our evidence promising because, for these samples, already with 55 seconds we can use what novelty we observe to deem sufficient progress in terms of environment-sensitive logic being satisfied. Overall, the default duration strikes a good balance between what we can observe in a run for our purpose and how we can make effective use of the limited budget for all runs.

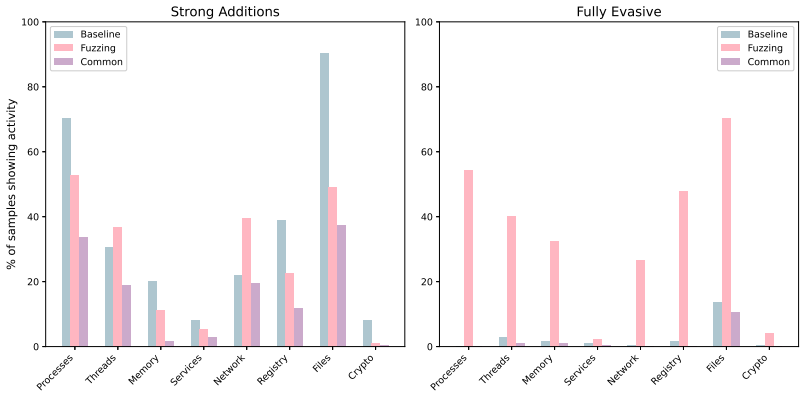
Internal Fuzzer Metrics. We now study how PFUZZER uses the time budget to attempt executions, generate policies, and advance exploration. Table 4 reports the various metrics we collected, averaging them for samples within each of the three groups considered for this RQ3 study.

The first element that we observe is that PFUZZER performs significantly more runs than the lower bound of 10 that the budget allows. This is mainly due premature execution termination, especially for samples in the *Fully evasive* groups, but also when PFUZZER triggers evasive tactics (absent, for the meaning of the group labels, in the baseline run) in samples from the two other groups. Less frequently, some samples conclude their work in less than the maximum run duration because they injected some process or spawned a child to continue execution.

We also note that, on average, PFUZZER retains as novel about 18 execution policies for samples in the first two groups, and about 29 for *Fully evasive* ones. Each policy captures new external or internal coverage compared to the plurality of trials attempted until its creation. For the first two groups, the policies account both for additional activity reached and for evasive paths PFUZZER unveils. The higher number for the third group comes from the fact that evasive tactics are typically multiple [2], [3] and may be interleaved with other activity (API calls or new code). Such activity, when unveiled, is precious for PFUZZER to retain the policy in the queue and use it in subsequent exploration to try and meet also the next batch of environmental dependencies through policy mutations.

Using external coverage on queried environmental information to drive mutations (Section 3.2) shows here as an effective choice: PFUZZER needs only a limited number of trials, on average about 4-6, before unveiling new coverage. This aspect is valuable because each trial is expensive (unlike software testing, we cannot do them in thousands), and reaching novelty fast is key for efficiency. To reach the configuration with most activity, PFUZZER takes on average about 16-18 trials with samples from the two groups with additions, and 29 with *Fully evasive* ones (higher likely due to the more complex expectations to meet). These numbers are way lower than the total number of runs, revealing that PFUZZER typically finds such configuration way before exhausting the time budget.

As for rules from policies, the table shows how many mutations PFUZZER encodes in the execution policy with most observed activity. The table distinguishes mutations according to the descriptions we gave in Section 3.4.



	Number of samples	
Behavior	*Strong add.*	*Fully evasive*
Communication	66	45
Data wiping	69	40
Encryption	2	7
Injection	6	42
Launching	91	55
Persistence	46	92
Spyware	8	14

Figure 2. Additional activities: type of operations (left) and manually identified behaviors (right) under the best execution policy w.r.t. baseline run.

Two facts stand out most: retargeting an operation is on average the most recurrent and therefore valuable action, and leaving (other) accesses to environmental information unaltered is also very important. The first suggests that many tactics may not depend only on a binary outcome for an access but also on its contents; also, retargeting seems even more valuable for samples that already show conspicuous activity in the baseline run. The second suggests that altering every query is often unnecessary, and we argue likely detrimental: it may trigger tactics that do not affect the current machine (as we saw for BLUEPILL in Section 5.2) or it may break the operation of the sample. Unlike existing systems, our approach can discriminate manipulations by their effects on execution through the coverage feedback and retain or discard them accordingly.

Environmental Artifacts. Besides how PFUZZER achieves its performance, an equally compelling question for defense architects is which aspects of the environment malware appears to be most sensitive to. We reviewed the logs to identify the most frequent query types and when PFUZZER altered execution behavior. Table 5 reports on samples with *Strong additions* and *Fully evasive* by grouping the most common query types that were altered in the execution policy that revealed the most activity. We focus on these two groups due to the wealth of unveiled behaviors. For both, the most frequent critical operations are checks on files, existence of registry paths, and time.

With *Fully evasive* samples, queries for VM-related conspicuous aspects included files, registry keys, GUI artifacts, processes, and hardware. We then witnessed how samples look up files and registry paths for anti-virus and monitoring software, and `ProductID` registry values linked to online sandboxes. Samples showed sensitivity also to aspects that sandbox users fine-tune manually, such as executable filename (e.g., presence of the `.exe` suffix) and path, working directory, keyboard layout, and installed programs. We spotted traits such as checking for Active Setup and Windows Defender settings (and eventually tampering with them for the latter) apparently missed in literature [2], [3]. Finally, we observed checks for files and registry keys that a prior-stage payload should create.

Samples with *Strong additions* often varied their tactics depending on conditions and components unrelated to evasion. The specific aspects being checked showed high heterogeneity. A noteworthy case involved checking for the presence of vulnerabilities (e.g., the `acsipc_server` named pipe) to enable privilege escalation or more stealthy operation. For many of these samples, we also observed in the logs queries typical of evasive tactics; however, as these specific tactics did not expose our baseline analysis environment, the samples showed malicious activity from the very first run.

	Fully evasive			*Strong additions*		
Category	**Rank**	**Samples**	**Avg #**	**Rank**	**Samples**	**Avg #**
File metadata/contents	1	88.36%	5.02	1	92.31%	5.78
Open registry key	2	76.03%	3.93	2	52.07%	4.24
Time/delay check	3	69.86%	1.57	3	63.31%	1.80
Query registry key	4	49.32%	1.90	7	21.89%	1.68
Processes	5	54.79%	1.26	5	51.48%	1.25
GUI elements	6	36.99%	1.74	6	22.49%	2.21
Mutex creation	7	45.89%	1.30	4	56.21%	1.46
Username	8	21.92%	1.31	9	8.88%	1.67
CPU details	9	16.44%	1.00	10	13.02%	1.00
Hostname	10	10.27%	1.40	8	20.71%	1.51

TABLE 5. TYPES OF QUERIES ALTERED IN THE POLICY WITH MOST ACTIVITY (COUNTS '#' ARE AVERAGED OVER AFFECTED SAMPLES).

For interested readers, in the companion GitHub repository of this paper we make available a list of the most relevant environmental artifacts we observed (and avoid an exhaustive description here for space limitations).

5.4. RQ4: Insights from Alternative Paths

Throughout this section, we seek to answer from multiple angles our last research question on the nature of the additional paths PFUZZER unveils: we study the same 457 samples as in the previous section, and separately the 344 *No extra activity* samples that looked non-evasive in RQ1 but, as we will learn, often have dormant evasive paths.

Due to space limitations, we refer to Appendix B for an analysis of code coverage variations along new paths.

Types of Activities. We start by looking into the types of new activities we observe. For samples from the *Mild additions* group, we mentioned in Section 5.1 that we observe manipulations involving predominantly files (113 samples) and the registry (37), and seldomly services (3).

For the other two groups, we collect more fine-grained indicators to characterize the new activities exposed. We partition in functional groups the externally observable operations that PFUZZER tracks and count for each group for how many samples we witness operations of this kind. The three per-group bars of Figure 2 include a sample,

respectively, if the baseline run shows operations from the group, if the best execution policy reveals new operations from the group, and if any operations from the group show in both (as opposed to them showing only when fuzzing).

The third bar is helpful to highlight a peculiarity for samples in the *Strong additions* groups. For them, PFUZZER often unveils within a group new operations that complement or replace others that show for that group in the baseline run. A "single-run" analysis solution would overlook these additional facets of a dimension, as they show based on environment-dependent conditions. For example, we found a sample that changes its handshake with its C&C server, and one that chooses between injection and process launching depending on the user account being in the admin group. A single-run analysis also trivially misses dimensions that never show in the baseline run: this happens for all the samples contributing to the fuzzing bar but not to the third one. This dynamic is particularly evident with the *Threads* and *Networking* groups, as we record activity of either kind for many more samples when fuzzing than in the baseline run.

This very dynamic is then dominant for the *Fully evasive* samples. This is expected, as these samples minimize their conspicuous activity when they detect an analysis environment in the baseline run, with the exception of file manipulations which are not necessarily malicious per se (that is, most programs manipulate files). Figure 2 shows how, with these samples, only through fuzzing we enable and in turn witness activity from the various groups.

We conclude the analysis by manually reviewing the recorded additional operations and seeing if we can attribute some with certainty to a typical malware behavior. Figure 2 shows the breakdown for the two groups. In the *Strong additions* group, the most recurring behaviors are launching (via process or service creation), data wiping, and remote communications. In the *Fully evasive* group, persistence is by far the most recurring behavior (this is expected, as evasive tactics are expected to safeguard such a conspicuous behavior), followed by launching, remote communications, and code injection; we remark how all these behaviors have a conspicuous malicious footprint.

Paths and Queries. To measure the alternative number of execution paths PFUZZER witnesses for a sample, we assemble the recorded external coverage activity into a graph (in general a forest, typically a tree in practice). We build one graph covering only the actions that bring about changes to the machine, and one where we include also the environmental information accesses PFUZZER observes.

With the first type of graph, we record a similar average number of unique alternative paths for the three sample groups: 7.76 with *Mild additions*, 6.46 with *Strong additions*, and 7.58 with *Highly evasive* ones. When we include the querying activity, the proportion changes: the number of paths is similar for the first two groups (respectively, 12.29 and 9.99), but almost twice as high for the third (22.4). The higher amounts of paths in all groups relates to the fact that a sample can exercise slight differences in the querying pathways before reaching a point where it alters system state. We explain the many more paths for *Fully evasive* samples with the existence of multiple stages of environment-sensitive checks, with each stage potentially introducing branching points in the execution. Single-run analysis solutions would miss this wealth of alternative, feasible executions for a sample.

Samples with No Extra Activity. We conclude by studying the 344 sample with conspicuous activity in the baseline run and no extra activity in PFUZZER. Their description may suggest that these samples may not be evasive: in reality, for 243 of them (70.64%), our method can expose "silent" evasive tactics, which make these samples valuable for our study. The remaining 100 can either be false positives from static classification or contain tactics that our analysis was not able to trigger in the experiment.

For these 243 samples, we identify on average 2.06 accesses to environmental information in the baseline run that, if altered in their course, steer the sample to evasion paths. Across all the paths we explore (7.43 paths per sample), early termination (1.39 paths) is more frequent than stalling tactics (0.88 paths). For an average of 96.02 external coverage events (either queries or changes to the system) we witness in the baseline run, we register after 15.22 such events the first access for which, as we altered it, we witnessed an evasion path. This is coherent with the common belief of evasive logic appearing already in (but not being limited to) the early stages of execution.

When analyzing PFUZZER's output for those samples, we identified two evasive techniques with no prior public record. The analysis we did for them became part of the popular Unprotect knowledge base of evasions [47]. One technique identifies VirtualBox VMs by scanning network shared folders through a resource enumeration instead of the well-established `WNetGetProviderName` API call to acquire conspicuous provider names. The other technique exposes VirtualBox disk drives by issuing a `SMART_RCV_DRIVE_DATA` I/O control code to the device driver, unlike classic serial number or model queries.

6. Conclusion

We have presented PFUZZER, a system embodying a novel approach to multi-path exploration for environment-sensitive malware. By proposing and implementing new fuzzing design techniques tailored around the specificities of malware, we have shown how coverage-guided fuzzing can be a promising way to analyze alternative behaviors from real-world Windows malware, outclassing the state of the art in academic research. These behaviors may not necessarily pertain only to evasive tactics, but often express a behavioral diversity and variability that past research observed or predicted for much general malware. We hope the ideas presented in this paper may stimulate new ideas and uses for our methodology. To favor such endeavors, we make available an annotated malware dataset.

Acknowledgments

We thank our anonymous reviewers for their feedback, Federico Palmaro for his help in the early stages of the project, Floris Gorter for sharing the evaluation materials for BLUEPILL from the ENVIRAL paper, and Cristiano Giuffrida for several stimulating conversations. This work has been partially supported by projects SERICS (PE00000014) and Rome Technopole (ECS00000024) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] M. Botacin, "Fuzzing and symbolic execution for multipath malware tracing: Bridging theory and practice via survey and experiments," *Digital Threats*, Oct. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3700147

[2] L. Maffia, D. Nisi, P. Kotzias, G. Lagorio, S. Aonzo, and D. Balzarotti, "Longitudinal study of the prevalence of malware evasive techniques," *CoRR*, vol. abs/2112.11289, 2021. [Online]. Available: https://arxiv.org/abs/2112.11289

[3] N. Galloro, M. Polino, M. Carminati, A. Continella, and S. Zanero, "A systematical and longitudinal study of evasive behaviors in windows malware," *Comput. Secur.*, vol. 113, no. C, Feb. 2022. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102550

[4] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 338–357. [Online]. Available: https://doi.org/10.1007/978-3-642-23644-0_18

[5] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 829–844.

[6] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro, "On the dissection of evasive malware," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2750–2765, 2020.

[7] F. Gorter, C. Giuffrida, and E. Van Der Kouwe, "Enviral: Fuzzing the environment for evasive malware analysis," in *Proceedings of the 16th European Workshop on System Security*, ser. EUROSEC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 8–14. [Online]. Available: https://doi.org/10.1145/3578357.3589455

[8] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: Vmm detection myths and realities," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07. USA: USENIX Association, 2007.

[9] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient Detection of Split Personalities in Malware," in *Proceedings of the 17th Symposium on Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[10] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: bare-metal analysis-based evasive malware detection," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 287–301.

[11] C. Song, P. Royal, and W. Lee, "Impeding automated malware analysis with environment-sensitive malware," in *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, ser. HotSec'12. USA: USENIX Association, 2012, p. 4.

[12] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 51–62. [Online]. Available: https://doi.org/10.1145/1455770.1455779

[13] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 386–395. [Online]. Available: https://doi.org/10.1145/2664243.2664252

[14] Z. Deng, X. Zhang, and D. Xu, "Spider: stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 289–298. [Online]. Available: https://doi.org/10.1145/2523649.2523675

[15] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. USA: IEEE Computer Society, 2015, p. 55–69. [Online]. Available: https://doi.org/10.1109/SP.2015.11

[16] G. Pék, B. Bencsáth, and L. Buttyán, "nether: in-guest detection of out-of-the-guest malware analyzers," in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/1972551.1972554

[17] M. Brengel, M. Backes, and C. Rossow, "Detecting hardware-assisted virtualization," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 207–227. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_11

[18] D. C. D'Elia. My ticks don't lie: New timing attacks for hypervisor detection. [Online]. Available: https://hdl.handle.net/11573/1499636

[19] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "Goldeneye: Efficiently and effectively unveiling malware's targeted environment," in *International Symposium on Recent Advances in Intrusion Detection*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:2138226

[20] W. You, Z. Zhang, Y. Kwon, Y. Aafer, F. Peng, Y. Shi, C. Harmon, and X. Zhang, "Pmp: Cost-effective forced execution with probabilistic memory pre-planning," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1121–1138.

[21] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Rambo: Run-time packer analysis with multiple branch observation," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 186–206. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_10

[22] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, ser. SP '07. USA: IEEE Computer Society, 2007, p. 231–245. [Online]. Available: https://doi.org/10.1109/SP.2007.17

[23] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 143–163. [Online]. Available: https://doi.org/10.1007/978-3-540-70542-0_8

[24] M. Ollivier, S. Bardin, R. Bonichon, and J.-Y. Marion, "Obfuscation: where are we in anti-dse protections? (a first attempt)," in *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*, ser. SSPREW9 '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3371307.3371309

[25] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:85546717

[26] Dynamorio. [Online]. Available: https://dynamorio.org/

[27] Intel. Pin - a dynamic binary instrumentation tool. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[28] D. C. D'Elia, S. Nicchi, M. Mariani, M. Marini, and F. Palmaro, "Designing robust api monitoring solutions," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 392–406, 2023.

[29] D. C. D'Elia, L. Invidia, F. Palmaro, and L. Querzoni, "Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts," *Digital Threats*, vol. 3, no. 2, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3478520

[30] Malicious apis. [Online]. Available: https://malapi.io/

[31] C. Rossow, C. Dietrich, and H. Bos, "Large-scale analysis of malware downloaders," in *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 42–61. [Online]. Available: https://doi.org/10.1007/978-3-642-37300-8_3

[32] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitras, "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, Aug. 2021, pp. 3487–3504. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/avllazagaj

[33] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[34] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 285–296. [Online]. Available: https://doi.org/10.1145/2046707.2046740

[35] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-Fuzzing techniques," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1913–1930. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/jung

[36] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "AntiFuzz: Impeding fuzzing audits of binary executables," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1931–1947. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/guler

[37] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed)," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ser. Asia CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 15–27. [Online]. Available: https://doi.org/10.1145/3321705.3329819

[38] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero, "Measuring and defeating anti-instrumentation-equipped malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 73–96.

[39] M. S. Karvandi, M. Gholamrezaei, S. Khalaj Monfared, S. Meghdadizanjani, B. Abbassi, A. Amini, R. Mortazavi, S. Gorgin, D. Rahmati, and M. Schwarz, "Hyperdbg: Reinventing hardware-assisted debugging," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1709–1723. [Online]. Available: https://doi.org/10.1145/3548606.3560649

[40] C. Assaiante, S. Nicchi, D. C. D'Elia, and L. Querzoni, "Evading userland api hooking, again: Novel attacks and a principled defense method," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 21st International Conference, DIMVA 2024, Lausanne, Switzerland, July 17–19, 2024, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 150–173. [Online]. Available: https://doi.org/10.1007/978-3-031-64171-8_8

[41] Yara. Anti-debugging and anti-virtualization yara rules. [Online]. Available: https://github.com/Yara-Rules/rules/blob/master/antidebug_antivm/antidebug_antivm.yar

[42] Detect-It-Easy. A portable executable analyzer tool. [Online]. Available: https://github.com/horsicq/Detect-It-Easy

[43] Avclass. Avclass - a tool for malware labeling. [Online]. Available: https://github.com/malicialab/avclass

[44] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *International Symposium on Recent Advances in Intrusion Detection*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:7715150

[45] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 1009–1024, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:12665057

[46] A. Kuechler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, "Does every second count? time-based evolution of malware behavior in sandboxes," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231798972

[47] J.-P. L. Thomas Roccia. Unprotect project. [Online]. Available: https://unprotect.it/

Appendix A. Evaluation Issues of Enviral

This section discusses issues that we found in the method the authors of ENVIRAL used for their preliminary evaluation for comparing it against BLUEPILL. The evaluation included 338 malware samples from 2019 obtained from the VirusTotal Academic collection and very likely to be evasive. We attempted a similar experiment using the 220 samples that we chose with our dataset construction method from the 2019 archive from the VirusTotal Academic collection. Coherently with the setup of the ENVIRAL paper, we run the tests in a Windows 7 VM.

The authors modified BLUEPILL to collect invocations of noisy APIs (for querying the machine or for altering its state, with no distinction in importance) and compute the increment in such activity compared to a baseline run with the anti-evasion tactics of BLUEPILL disabled. This choice enables a direct comparison with ENVIRAL, as the system focuses on evolving a single path that brings an increase of such activity compared to the baseline run.

For reproducing their experimental workflow, we used the version of BLUEPILL modified by ENVIRAL's authors. We identified three main issues in their work:

1) a clerical error when modifying BLUEPILL for intercepting execution termination;
2) an unsound choice of the time budget for BLUEPILL to offset DBI overheads;
3) an imperfect evaluation metric that is also prone to distortion effects.

These issues explain why the results we presented for ENVIRAL in the RQ2 experiments are worse than those for BLUEPILL, and make us conclude that the authors incorrectly claimed an improved efficacy over BLUEPILL.

Clerical Error. The authors modified BLUEPILL's original code to terminate execution whenever a Windows exception was triggered and the exception handler of Pin caught it. However, this approach is problematic because not all exceptions result in execution failures: with malware, some may be known anti-debugger techniques. Additionally, BLUEPILL has specific mechanisms in place to handle certain exceptions. But most importantly, the original exception handler of BLUEPILL leaves the sample under test to manage such exceptions, which is what anti-debugger techniques expect on a normal execution. By prematurely terminating execution for BLUEPILL, a significant amount of observable activity is lost.

Unsound Budget. The authors also modified BLUEPILL to execute each sample for a maximum of 10 seconds (and

Enviral	Bluepill	Settings		Geomean for activity (as in [7])			Per-sample head-to-head activity count		
Time (s)	Time (s)	Verbose	Error	BluePill	Enviral	Increase (%)	BP > Enviral	Enviral > BP	Equal
2.5	10	Off	Yes	1.07	1.14	6.54	85	54	81
2.5	10	Off	No	1.03	1.14	10.32	61	52	107
2.5	15	On	Yes	0.89	1.14	28.54	146	35	39
2.5	15	On	No	1.14	1.14	0.10	193	7	20
10	30	Off	Yes	1.13	1.23	8.67	114	47	59
10	30	On	No	1.29	1.23	-4.77	185	15	20
25	120	Off	Yes	1.14	1.25	9.94	104	58	58
25	120	On	No	1.24	1.25	0.84	182	15	23

TABLE 6. ANALYSIS RESULTS FOR BLUEPILL AND ENVIRAL ON THE 220 SAMPLES FROM 2019 IN OUR DATASET

up to 15 seconds when in a verbose mode not used for the evaluation). Unfortunately, these time limits are not ideal for a DBI-based solution, which faces higher overheads in the very first seconds of executions (i.e., bringing up the just-in-time compilation process and filling the code cache with the firstly executed functions of the samples). We noticed how it was sufficient to increase the budget by a few seconds to make it appear in BLUEPILL's logs all the additional activity that ENVIRAL showed for many samples: activity due not to improved anti-evasion capabilities, but to overhead/timing distortions. We experimented (Table 6) with different budgets to establish a more fair and sound comparison of the capabilities of the two systems, and these tests resulted in rather close values for the two systems for the metric proposed by the authors.

Imperfect Metric. The evaluation metric of ENVIRAL is the geometric mean of the increase in activity relative to a baseline execution, with the ratio computed for each sample with respect to the baseline run of the respective system. However, focusing on the total volume of activity is not accurate: for example, an increase in activity could be due to adversarial techniques designed to steer the analysis towards dead-end paths. We also noted that the implementation did not differentiate the point of origin for such activity between the instructions of the samples and the implementation of Windows libraries. We also identified some redundancy in the counts, for example including both `CreateProcessInternalW` and its internal call to `NtCreateUserProcess` when intercepting process creation. The metric we use in this paper focuses instead on conspicuous activity that brings about transient or permanent effects to the machine and that originates from the code of the sample only.

Experimental Results. Table 6 shows the results of our experiments on the 220 samples from our dataset that we have for 2019. We plot the different settings (time budget per run, use of the verbose mode for logging activity in BLUEPILL, presence of the clerical error) we use for the configurations and two metrics: the one as in the ENVIRAL paper, and a variant of it where we simply count on each sample which of the two systems revealed more activity (i.e., a difference of at least 5% in the count) in terms of APIs that ENVIRAL deems as important. The results show that 25 seconds is the budget that leads to the highest geomean activity increase under ENVIRAL, which justifies our budget choice in the RQ2 evaluation. They also show the improvement over BLUEPILL is modest in that setting (1.25 vs 1.24), whereas the API counts per sample show that the metric is highly imprecise and skewed: while it suggests an improvement, only on 15 samples ENVIRAL finds more activity than BLUEPILL, which instead prevails for 182 and matches ENVIRAL for 23. For this reason, we do not use ENVIRAL's metric for this paper's experiments.

With our metric for assessing conspicuous activity, we note neither system finds relevant activity for 39.09% of the samples. When at least one does, they are equally matched on 13.64% of the samples, BLUEPILL prevails on 32.72% of them, and ENVIRAL on 14.55% of them.

Appendix B. Additional Evaluation Materials

This section features charts and descriptions that we do not include in the main evaluation due to space limitations.

Evaluation Metric. As we mentioned, our goal is to enable a semi-qualitative assessment of a sample's activity, as opposed to relying on uninformed API counts as in ENVIRAL [7] (a slippery road, as we discussed in the previous section) or to manual labeling of activity logs as in BLUEPILL [6] (expensive and difficult to replicate).

As part of its external coverage feedback, PFUZZER tracks the identity and callsite for 146 APIs that alter system state. During the experimental comparison of RQ2, we added analogous instrumentation to BLUEPILL and ENVIRAL to track the invocation of these APIs. In the execution logs we obtain for each execution in one of the systems (one for BLUEPILL, many for the other two), we examine the first occurrence of any transient or permanent system changes, distinguishing APIs by their call site.

As a first indicator, we focus on a subset of these 146 APIs that are most strongly relatable to malicious activity. For instance, for APIs that pertain to system service management, we prioritize APIs related to the creation or initiation of new services, disregarding those, for example, available for querying service status.

Next, we try to see if we can attribute with confidence certain system changes to high-level malware behaviors (e.g., injection, launching, networking, spying). We take into account API sequences (e.g., for injection, writing to a region and executing from there), parameters (e.g., known registry paths commonly abused for persistence), and recurrence (e.g., cryptographic primitives for ransomware).

The output of this process is, for each run, a list of the activities we recognize throughout the execution. We can use these lists to directly compare runs of the same system, as when attributing sample labels in the RQ1 analysis for the additional activities unveiled, as well as of different systems, as we did in the RQ2 analysis. We make available the code (and logs) for the evaluation metric in the companion GitHub repository of this paper.

Code Coverage. To look into what code different paths execute, in Figure 3 we measure code coverage variations between the baseline run and the one with the execution

		Step 1	Step 2			Step 3	Step 4	Step 5
Year	**Initial**	**Vhash**	**Yara**	**Skip dbg**	**Systems**	**.NET**	**Families**	**Dynamic**
2018	64 313	18 105	12 043	4 042	586	499	306	281
2019	59 104	12 989	7 588	2 854	510	480	315	295
2020	52 512	16 829	3 555	1 175	203	196	109	104
2021	25 176	6 189	3 841	1 946	362	354	242	213
2022	65 919	8 555	6 261	1 260	186	169	153	97
2023	48 207	8 484	6 901	1 064	171	157	137	88
Total	**315 231**	**71 151**	**40 189**	**12 341**	**2 018**	**1 855**	**1 262**	**1 078**

TABLE 7. DETAILED BREAKDOWN OF THE DATASET CONSTRUCTION PROCESS

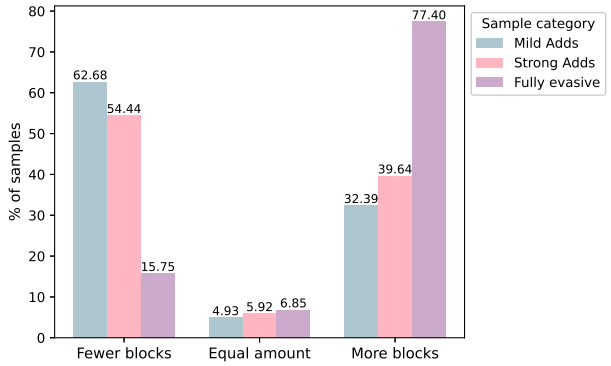


Figure 3. Distinct basic blocks that the sample traverses under the first execution policy unveiling new conspicuous activity w.r.t. baseline run.

policy that makes PFUZZER witness new conspicuous activity for the first time. We consider the unique basic blocks a run traverses and differentiate samples by groups.

For samples with *Mild additions* (142), novelty occurs on an execution path that traverses fewer distinct basic blocks than in the baseline run in 62.68% of the cases (89 samples). This often relates to the sample engaging in activities alternative to some of those from the baseline run; alternatively, execution may interleave these new activities with those showing in the baseline run, but the fuzzing run times out before the sample can complete them all. PFUZZER is sensitive to these differences as it reasons on the identity (API name and call site) of coverage entities for assessing novelty in the feedback. In rare cases (7 samples), the blocks are numerically the same but the identity of some (and possibly the dataflow at others) change. In the remaining 32.39% of the cases (46), novelty comes with a higher count of traversed blocks.

For samples with *Strong additions* (169), the trends are quite similar and our interpretation is analogous to above, but the prevalence of cases with higher counts is slightly higher, totaling 39.64% of the cases (67 samples).

Fully evasive samples (146) show a different dynamics, with higher amounts of traversed blocks as the most common case (77.40%). This trend is expected, as we assume these samples tend to terminate execution early or stall it and, by the definition we gave in RQ1 for the set, the baseline run triggers their evasive tactics.

Appendix C. Dataset Details and Inconclusive Samples

Breakdown. Table 7 provides a detailed breakdown of the dataset construction process we presented in Section 4.

Inconclusive Samples. We then provide more details for samples that we termed *Inconclusive* in Section 5.1. As discussed in the paper, most of these samples present orthogonal challenges that are beyond the capabilities of current approaches. We hope to ease future work in the area by sharing these samples and our analysis notes for them. These challenges are (see Table 1 for the set counts):

- *Command-line arguments*: samples that demand specific command-line arguments for execution. Using a debugger, we provided arguments obtained through manual reversing work and observed how this steered execution to sections of code where interesting behaviors occurred. Some samples accepted a range of arguments, each leading to different behaviors based on the input.
- *External dependencies*: samples requiring specific items, such as DLL files (e.g., `steam_api.dll`, `lib_game.dll`), specific resources not present in the PE file, a valid network connection (our simulator was insufficient), or files with particular content (mainly, configuration like `setup.cfg`).
- *Interactive behavior*: These samples include three behaviors: installers that initiate an installation process requiring GUI interactions, samples that trigger malicious activity after one or more user interactions (e.g., message boxes), and samples that display a GUI for text input (e.g., keygens).
- *Program failures*: These samples failed to execute in our analysis environment due to various reasons, including faulty applications that triggered error messages, software unable to run in Win32 mode, or cases where the OS indicated the application was incompatible with the system.

On a related note, samples that we dubbed *Undetermined evasions* exhibit evasive behavior, such as terminating after executing only a few instructions or without displaying malicious activity. However, we were unable to identify with certainty the root cause of their evasion. Many of these cases appear to come from modified versions of executable protectors, especially Safengine.

Appendix D. Themida Samples and DynamoRIO's Defect

In our dataset construction process, we encountered 122 samples shielded by a particular configuration of the Themida executable protector that hit structural bugs in the implementation of DynamoRIO. While we reported these DBI cache coherency bugs to its maintainers (GitHub issue #6567), they have not been solved to date.

To validate our belief that these samples do not hit a methodological limitation of our method, we implement a

Category	Samples
Fully evasive (and countered)	89 (72.95%)
Undetermined evasions	16 (13.11%)
Unable to start program	13 (10.65%)
Interactive	2 (1.64%)
External dependencies	2 (1.64%)
Total	122

TABLE 8. THEMIDA-PROTECTED SAMPLES HITTING ONE OF TWO KNOWN CACHE COHERENCY BUGS IN DYNAMORIO.

small-scale version of it in Pin, targeting for fuzzing only the operations that we witness in a baseline run (more specifically, `cpuid` and two variants of `RegOpenKey`).

Table 8 presents the results of this analysis. For 72.95% of the samples, we successfully bypass the evasive techniques and uncover significant conspicuous behaviors according to our evaluation metric. However, 13.11% of the samples evade detection without revealing noteworthy behavior, likely due to the presence of additional evasive techniques for which further investigation is required. The remaining 13.99% of the samples are akin to the *Inconclusive samples* defined in RQ1 and further discussed in Appendix C: these samples are faulty executables or necessitate either external dependencies or user interaction.

These results make us conclude that also these samples could be handled by our reference implementation of PFUZZER once the DynamoRIO's bugs are solved. As this may require substantial engineering effort orthogonal to the purpose of this research, we left it to future work.