Terminator: Enabling Efficient Fuzzing of Closed-Source GUI Programs by Automatic Coverage-Guided Termination

Jonas Zabel
Secure Software Engineering
Fraunhofer SIT | ATHENE
Darmstadt, Germany
jonas.zabel@sit.fraunhofer.de

Philip Kolvenbach
Information Security
DZ Bank
Darmstadt, Germany
philip.kolvenbach@dzbank.de

Steven Arzt
Secure Software Engineering
Fraunhofer SIT | ATHENE
Darmstadt, Germany
steven.arzt@sit.fraunhofer.de

***Abstract*—When fuzzing a proprietary file-processing program, one typically executes the whole program repeatedly with sampled input files, and distinguishes between normal and abnormal termination. While this works well for many command-line utilities, it is more complicated for programs that usually do not terminate after input file processing. Many real-world applications are examples of such programs, in particular, those with a graphical user interface (GUI), such as image editors, media players and document viewers. In these cases, the fuzzer has to define the scope of the execution and forcefully terminate the program under test.**

In order to efficiently fuzz test file-processing programs with a GUI, a standard approach is to define a dedicated testing harness, which executes the file processing in isolation and strips irrelevant program parts. However, this either requires the source code of the program or an expert's effort in reverse engineering. Alternative approaches work on the unmodified binary of the program, and use a heuristic to decide when the input processing is likely done. For example, one can terminate the program after a fixed timeout or once its CPU usage has dropped below a threshold. We show that these heuristics, while simple to implement, are inefficient and ineffective.

We present TERMINATOR, a fully-automated approach to facilitate efficient fuzzing of closed-source file-processing programs with a GUI. TERMINATOR modifies the binary of the program under test so that it automatically terminates when code coverage stops increasing without user interaction. Consequently, TERMINATOR (1) ensures that the program terminates soon after the input processing instead of waiting for user interaction, and, at the same time, (2) prevents premature termination during input processing. We show that TERMINATOR outperforms the timeout and CPU usage heuristics and significantly increases fuzzing efficiency.

***Index Terms*—fuzzing, GUI, dynamic instrumentation, Windows, closed-source software**

I. INTRODUCTION

Fuzz testing, or simply fuzzing, is a well-established technique to automatically discover bugs in software. In essence, fuzzing repeatedly executes the program under test (PUT) using sampled inputs and monitors for abnormal behavior such as buffer overflows, dangling pointer dereferences, segmentation faults, memory leaks, or hangs. Since the original study on fuzzing for reliability testing by Miller et al. [35], it has become an important tool to find bugs and particularly vulnerabilities in software. Industry-leading software companies employ large-scale fuzz testing; Google alone reports tens of thousands of bugs discovered through fuzzing [23].

While fuzzing has seen tremendous success on Linux systems, effectively fuzzing Windows applications remains challenging. This is particularly concerning as Windows dominates the desktop market with over 70% share [39], making its applications prime targets for attackers. The Windows ecosystem presents unique obstacles that impede automated security testing: programs heavily rely on GUI interactions, most applications are closed-source, and the platform lacks efficient process creation primitives found in Unix-like systems [26].

Most research on fuzzing focuses on finding vulnerabilities in input file handling [27]. Under this threat model, the PUT opens an input file whose parsing and processing may be vulnerable, with no user interaction beyond opening the program. This works well for command-line utilities that process input and terminate. However, programs with graphical user interfaces (GUI) like image editors, document viewers, and media players complicate the process since they wait indefinitely for user interaction after displaying results.

Few strategies exist to fuzz test proprietary GUI applications effectively. Traditional approaches have significant limitations. One common method involves the manual creation of testing harnesses by distilling the input processing of the PUT into a dedicated testing harness [24, 29]. However, this approach requires extensive reverse engineering for closed-source applications [2], making it time-consuming and expertise-intensive. Another strategy involves automated fuzz-driver generation, which has seen success in tools like Winnie [26] for binaries. However, these tools can face challenges with complex applications where parsing logic is tightly coupled with the GUI, often requiring manual intervention to create a harness. For instance, Winnie's harness generation was primarily demonstrated on library functions (DLLs) and does not work with parsing logic embedded within the main executable, which is a common pattern in the applications we target. A third approach relies on heuristic-based termination using timeouts or CPU monitoring [32] to determine when input processing is complete, but this method is both inefficient and unreliable, often leading to premature termination or unnecessary waiting periods that slow down the fuzzing process.

We present TERMINATOR, a novel system that enables efficient automatic fuzzing of Windows GUI applications by modifying binaries to terminate once input processing is complete. Our key insight is that we can identify the logical end of input handling by analyzing code coverage patterns. Using dynamic binary instrumentation, TERMINATOR collects execution traces to identify functions involved in input processing, analyzes coverage data to determine optimal termination points that maximize coverage while preventing premature exits, and patches the binary to automatically terminate at these points, eliminating the need for user interaction.

This approach provides several advantages over existing solutions: it works directly on binaries without source code or harness development, does not require manual reverse-engineering of the PUT, achieves high efficiency by avoiding generic heuristic-based approaches, and strongly reduces per-application setup effort and expertise compared to other automated driver generation tools. In comparison to timeout-based approaches, TERMINATOR can terminate the program once it has finished processing *the file at hand* without having to wait for a timeout that is suitable for *all possible input files*.

Our main contributions are:

- TERMINATOR, a fully-automated approach to facilitate efficient fuzzing of proprietary file-processing programs with a GUI. TERMINATOR modifies the binary of the PUT so that it automatically terminates when the code coverage will not increase without user interaction. TERMINATOR is not a fuzzer, but a preprocessor that modifies a program before the actual fuzzing.
- A robust implementation that works across diverse file processing applications
- Empirical evidence that TERMINATOR can significantly improve fuzzing efficiency compared to traditional CPU monitoring and static timeout approaches.

This paper is organized as follows: Section II provides background on Windows fuzzing challenges. Section III details our coverage-guided termination approach. Section IV presents out evaluation methodology and Section IV presents evaluation results. We conclude in Section VII.

II. BACKGROUND AND RELATED WORK

Fuzz testing has received vivid interest by the software security research community [27, 28]. Most research focuses on improving input generation, commonly utilizing feedback obtained from previous inputs to steer the generation of new inputs towards unexplored paths; the feedback can be based on code coverage [46], branch coverage [13], data flow and taint tracking [14, 21, 31], control flow [38], projected security impact [15, 43], or even human clues [3].

Directed fuzzers attempt to craft inputs destined to reach specific code paths, such as new code paths or those affected by a particular change in the code base. This is often achieved through symbolic execution [11, 16, 22, 30, 37, 45, 47], and more recently by directed grey-box techniques, such as input-to-state correspondence or heuristic optimization methods aimed at minimizing a distance measure [5, 9, 12]. An orthogonal approach leverages input format grammars, which can be manually supplied or automatically inferred, to bypass early syntax checks and explore deep paths [4, 7, 25, 36, 41, 42].

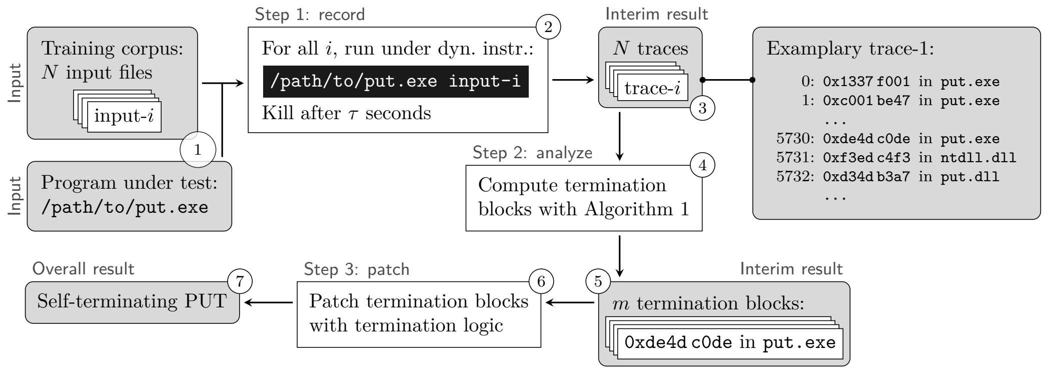
Despite these advances, most fuzzing research still focuses primarily on command-line utilities that terminate after processing input or shared libraries. Pure symbolic execution fuzzers such as KLEE [11] might theoretically handle non-terminating programs, but in practice they scale poorly with complex applications [8].

This challenge is acutely evident in the context of GUI applications, which are inherently non-terminating and event-driven. Fuzzing these applications has therefore been addressed through only a few key strategies that circumvent their persistent runtime:

Automated Harness Generation. As discussed in Section I, tools like Winnie's harnessgen [26] automate the generation of fuzzing harnesses to bypass the GUI and test parsing logic in isolation. While effective for well-structured libraries, this approach struggles with monolithic executables where parsing and GUI logic are tightly intertwined. Furthermore, Winnie's process cloning mechanism does not support WinAPI UI calls in cloned processes, limiting its applicability to applications that can be successfully decoupled from their UI.

Environment Fuzzing. EnvFuzz [33] introduces program environment fuzzing, which records and deterministically replays system call interactions to handle non-determinism in GUI applications. While a powerful technique, EnvFuzz has several limitations. First, it is designed for Linux and relies on intercepting a manageable number of libc syscalls; porting this to Windows, with its vast and poorly documented set of over 2000 syscalls in `ntdll` and `win32k`, would be a formidable engineering challenge. Second, EnvFuzz's recording phase requires a human-in-the-loop to define the end of an interesting interaction. Third, when program behavior diverges from the recording, EnvFuzz falls back to syscall emulation, which can introduce severe inaccuracies.

Heuristic-Based Termination. The most practical approach uses heuristics such as CPU idle monitoring or timeout-based termination [32] to determine when input processing is complete. CPU idle monitoring offers a practical approach by observing the application's computational activity over time. When an application transitions from actively processing input to waiting for user interaction, its CPU usage typically drops to near-idle levels. However, naive CPU monitoring approaches face several challenges: applications may have background threads that maintain low but consistent CPU usage, different applications exhibit varying idle patterns, and system noise can create false signals. Despite these complexities, CPU idle detection remains a widely applicable technique because it requires no application-specific knowledge and works across different software architectures. Fixed timeout approaches are simpler but inherently inefficient, wasting time on simple inputs



```
/path/to/put.exe
```

```
/path/to/put.exe input-i
```

```
   0: 0x1337f001 in put.exe
   1: 0xc001be47 in put.exe
      ...
5730: 0xde4dc0de in put.exe
5731: 0xf3edc4f3 in ntdll.dll
5732: 0xd34db3a7 in put.dll
      ...
```

```
0xde4d c0de in put.exe
```

Fig. 1: High-level overview of TERMINATOR. *"dyn. inst."* is an abbreviation for dynamic instrumentation.

while potentially truncating processing of complex inputs.

Despite the existence of these techniques, a significant gap remains in making complex, closed-source GUI applications efficiently "fuzzable" with minimal manual effort. This paper bridges this gap by introducing an automated technique for intelligent termination that enables accurate fuzzing of GUI applications without requiring extensive manual analysis or modification.

III. TERMINATOR: COVERAGE-GUIDED AUTOMATIC TERMINATION OF PROGRAMS

TERMINATOR modifies binaries of file-processing programs, such that they automatically terminate when the code coverage is not likely to increase without user interaction. Thereby, TERMINATOR facilitates efficient and effective fuzz testing of proprietary file-processing applications with a GUI, because it ensures that the program terminates soon after the input processing, instead of needlessly waiting for user interaction.

In our model, single-threaded GUI-based programs are launched at a time t_0. They start processing the input file at a time $t_p > t_0$. Eventually, they wait for user interaction at time $t_u > t_p$. In between, at times we denote t_x and t_y, the PUT may perform additional tasks unrelated to the input file such as checking for software updates. This gives us a time sequence $t_0 < t_x < t_p < t_y < t_u$. Intuitively, the PUT should be terminated at t_u the latest or, even better, at t_y. TERMINATOR is based on the insight that termination should happen when, given enough inputs, the same basic blocks are executed for every input. In other words, we want to terminate the PUT when it is no longer input-dependent. Since parsing is always input-dependent, this oracle approximates t_y.

We divide the operation of TERMINATOR into three stages:

Step 1 (record): Utilizing dynamic instrumentation, record control-flow traces of the PUT, i.e., complete execution histories, for input files from a training corpus.

Step 2 (analyze): Analyze the traces in order to select a set of basic blocks *(termination blocks)* from the PUT that appear as late as possible in all recorded traces.

Step 3 (patch): Modify the binary of the PUT to terminate whenever one of the termination blocks is executed.

See Figure 1 for an overview of TERMINATOR. We rely primarily on dynamic program analysis to examine the PUT because it is a hard to extract the control-flow graph from a binary alone [34]. In absence of a control-flow graph, many powerful static analysis methods are not applicable.

A. Notation

We denote the training corpus by $I = \{1, \ldots, N\}$, where $N \in \mathbb{N}$ is the total number of files and $i \in I$ are references to the individual files. We assume that we can instantiate the PUT such that it processes a specified input file; for many programs it suffices to pass the path to an input file as the first command-line parameter. For ease of presentation, we further assume that the PUT is single-threaded (see discussion in Section III-D3).

We define a basic block as a sequence of CPU instructions that starts with the target of a control-transfer instruction and ends with a control-transfer instruction. Furthermore, each basic block is located in a certain binary file at a certain position, the combination of which can be used to uniquely identify it; self-modifying and just-in-time compiled code is not addressed here. We denote the set of all basic blocks by $\mathbb{B}$ and use $\mathbb{B} = \mathbb{N}$ without loss of generality (the set of basic blocks is countable).

If we instantiate the PUT with input file $i \in I$ and let it execute for a fixed time $\tau > 0$, the control-flow trace, i.e., the complete ordered history of basic blocks it executes, can be written as the tuple

$$T_i = (t_i^1, t_i^2, \ldots, t_i^{n_i}) \in \mathbb{B}^{n_i},$$

for some $0 < n_i \in \mathbb{N}$. We assume that exactly one trace is collected per input file to keep the formalism simple; multiple traces per file can be handled without changing the formalism by including multiple copies of each file in the training corpus. The *trace set* of basic blocks in T_i is denoted by

$$S_i = \{t_i^1, t_i^2, \ldots, t_i^{n_i}\}, \tag{1}$$

and their union by $S = \bigcup_{i \in I} S_i$. To each basic block $x \in S_i$ we associate the position where it first appears in T_i, i.e., the *premiere*

$$p_i(x) = \min_k \{\, 1 \leq k \leq n_i : x = t_i^k \,\}.$$

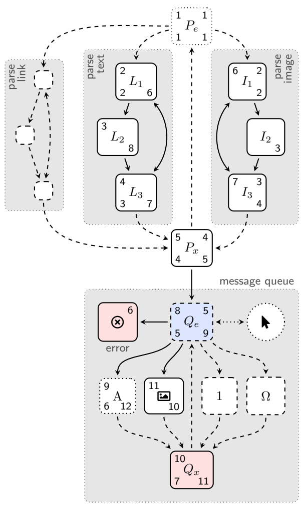


Fig. 2: Control-flow graph of a hypothetical program. The numbers in the corners of the nodes are the premiere orders and correspond to the o columns in Table I.

o_1	r_1	T_1	o_2	r_2	T_2	o_3	r_3	T_3	o_4	r_4	T_4
1	9 %	P_e	1	17 %	P_e	1	8 %	P_e	1	14 %	P_e
2	18 %	L_1	2	33 %	I_1	2	17 %	I_1	2	29 %	L_1
3	27 %	L_2	3	50 %	I_3	3	25 %	I_2	3	43 %	L_3
4	36 %	L_3	4	67 %	P_x	4	33 %	I_3	4	57 %	P_x
5	45 %	P_x	5	83 %	Q_e	5	42 %	P_x			P_e
		P_e	6	100 %	$\otimes$			P_e			L_1
		L_1				6	50 %	L_1			L_3
		L_4				7	58 %	L_3			P_x
		P_x						P_x	5	71 %	Q_e
		P_e						P_e	6	86 %	A
6	55 %	I_1						L_1	7	100 %	Q_x
7	64 %	I_3				8	67 %	L_2			Q_e
		P_x						P_x			…
8	73 %	Q_e				9	75 %	Q_e			
9	82 %	A				10	83 %	🖼			
10	91 %	Q_x				11	92 %	Q_x			
		Q_e				12	100 %	A			
11	100 %	🖼						Q_e			
		Q_x						…			
		Q_e									
		…									

TABLE I: Traces T_i of four input files through the graph in Figure 2. The r and o columns are the (relative) premiere orders.

Using absolute value bars $|\{\ldots\}|$ to denote the cardinality of a set $\{\ldots\}$, the number of distinct basic blocks in T_i up to position k is $|\{t_i^1, \ldots, t_i^k\}|$. Hence, we define the *premiere order* of $x \in S_i$ in T_i as

$$o_i(x) = |\{t_i^1, \ldots, t_i^{p_i(x)}\}|. \tag{2}$$

As an example, if we had a trace $T_i = (b, a, a, c, b)$, the premiere orders would be $o_i(a) = 2$, $o_i(b) = 1$ and $o_i(c) = 3$. Lastly, the *relative premiere order* of $x \in S_i$ in T_i is the ratio

$$r_i(x) = \frac{o_i(x)}{|S_i|}, \tag{3}$$

which takes values in $(0, 1]$. In the above example, we would have $r_i(a) = 2/3$, $r_i(b) = 1/3$ and $r_i(c) = 3/3$.

If we were to modify the semantics of a basic block x such that the program would terminate upon execution of x, *ceteris paribus*[1], the traces T_i' and trace sets S_i' of the modified program would be truncated versions of the original traces T_i and trace sets S_i. It is straightforward to see that $r_i(x) = |S_i'| \,/\, |S_i|$ whenever $x \in S_i'$. Therefore, we also call $r_i(x)$ the *trace coverage* in trace i at x. In our example, if we terminated the program at basic block b, only $r_i(b) = 1/3$ of the blocks in the original trace would be executed; potential vulnerabilities in a or c would be impossible to detect dynamically.

Taking all traces into account at once, the guaranteed trace coverage of x is the minimum

$$g(x) = \min\,\{\, r_i(x) : i \in I \text{ with } x \in S_i \,\}, \tag{4}$$

which is well-defined for all x in the trace set union S. Here, *guaranteed* is to be understood relative to the training corpus, not in general terms. For example, with $T_1 = (a, b, c)$ and $T_2 = (a, c, b, d)$ we have $r_1(c) = 3/3$ and $r_2(c) = 2/4$, so that termination at c guarantees a trace coverage of $2/4$. Analogously, if we terminate the PUT at any of a set $\{\} \neq X \subseteq S$ of basic blocks, the guaranteed trace coverage of X is

$$G(X) = \min\,\{\, g(x) : x \in X \,\}, \tag{5}$$

which is the lowest trace coverage at any basic block in X in any trace of the training corpus files.

[1]This assumes that multiple executions with the same input lead to the same control-flow trace, i.e., it ignores non-determinism.

B. Illustration of Terminator

To illustrate the approach, we consider a hypothetical PDF viewer with a simplified control flow like that illustrated in Figure 2, where the nodes represent basic blocks. The control-flow graph is only for illustration purposes – Terminator does not have such information. In this example, we analyze the program with a training corpus of four input files. The individual traces are shown vertically in Table I, in the columns T_1 to T_4. The premiere order (see eq. (2)) from columns o_1 to o_4 is annotated at the corners inside the nodes in Figure 2. The relative premiere orders (see eq. (3)) are in columns r_1 to r_4.

The basic blocks that are discovered last in the individual traces are $X = \{\otimes, \text{A}, 🖼, Q_x\}$. We have $G(X) = r_3(\text{A}) = 83\,\%$.[2] If instead we used the blocks $X' = \{\otimes, Q_x\}$, which are colored red in the figure, we had $G(X') = r_1(Q_x) = 91\,\%$. Since $G(X') > G(X)$, the choice X' would lead to an increased minimum proportion of basic blocks being executed, even though it does not contain all last-discovered basic blocks.

[2]For simplicitly, we assume here that the termination happens at the end of the termination blocks. In reality, we replace these blocks by entirely different code, see Section III-D.

The simple example illustrates that the partial orders the execution traces (T_1, …) define on the control-flow graph by virtue of the relative premiere orders are not necessarily compatible, so that they cannot be combined to find basic blocks that universally are discovered last. Therefore, TERMINATOR takes a different approach and seeks a set $X \subseteq \mathbb{B}$ of basic blocks that, among all sets that have a non-empty intersection with all traces, guarantees the highest relative premiere order across all traces, i.e., maximizes the minimum trace coverage $G(X)$. In this sense, the set X' is the optimal solution to our example problem and, hence, the set that TERMINATOR defines as the termination blocks.

C. *Formal description of* TERMINATOR

We denote by $U \subseteq \mathbb{B}$ the set of basic blocks that TERMINATOR is allowed to patch with termination logic. We call a subset $X \subseteq S$ a candidate solution, or feasible, if the blocks in X are allowed, i.e.,

$$X \subseteq U, \tag{6}$$

and X covers all recorded traces, in the sense that each trace contains at least one basic block of X, i.e.,

$$S_i \cap X \neq \{\} \quad \text{for all } i \in I. \tag{7}$$

Condition (7) ensures that the PUT would be terminated for all input files from the training corpus, if the PUT was modified to terminate at the blocks in X.

Among all feasible $X \subseteq S$, TERMINATOR looks for one that maximizes the objective function

$$\begin{aligned} f\colon \mathcal{P}(S) &\to \mathbb{R}^2, \\ X &\mapsto \big(G(X),\, -|X|\big), \end{aligned} \tag{8}$$

where $\mathcal{P}(S)$ is the power set of S. Here, the maximum is to be understood in terms of the usual lexicographic order on real tuples. Hence, the objective function always prefers a candidate solution X with a higher value $G(X)$, and only considers the cardinality of the solution as a tie breaker, i.e., if $G(X) = G(X')$ and $|X| < |X'|$ then X is preferable. Formally, the TERMINATOR problem is to find the solution $X \subseteq S$ of

$$\max_{X \subseteq S} \{\, f(X) : X \text{ fulfills (6) and (7)} \,\}. \tag{9}$$

The search space $\{\, X \subseteq S : X \text{ fulfills (6) and (7)} \,\}$ might be empty, because the set U is too restrictive. If this is the case, TERMINATOR fails with an error message.

Problem (9) can be understood as a generalized set cover problem, one of the classical NP-hard combinatorial optimization problems [17]. Here, the trace index set I shall be covered indirectly by a set $X \subseteq S$ of basic blocks through the membership index sets $\{\, i \in I : x \in S_i \,\}$, for $x \in X$; among all such covers $X \subseteq S$, the smallest one (cardinality-wise) that maximizes $G(X)$ is sought. It can be formulated as a mathematical mixed-integer linear program (MILP) and solved with specialized tools such as SCIP [20]. However, experiments have shown that this general and exact approach can take very long (longer than a day) in our case. Therefore, TERMINATOR currently uses a greedy algorithm to find an approximate solution to problem (9). While it cannot be guaranteed that the greedy algorithm finds the optimal solution, it is well known that greedy algorithms perform well in set-cover contexts [17].

Algorithm 1: Overview of TERMINATOR

```
   Data: Paths $p_1, \ldots, p_N$ to training corpus files, path $q$
         to executable of PUT, timeout $\tau > 0$, restriction
         $U \subseteq \mathbb{B}$
   Result: Patched, self-terminating executable
   Step 1 (record):
1  $T_i = \mathrm{recordControlFlowTrace}(q, p_i, \tau)$, for $i = 1, \ldots, N$
   Step 2 (analyze):
2  $A_0 = S \cap U$
3  $X_0 = I_0 = \{\}$
4  $k \leftarrow 0$
5  while $|I_k| < N$ do
6      if $A_k = \{\}$ then
7          Error "No solution exists"
8      end
9      $f_k(x) = \big(g(x),\ |\{\, 1 \le i \le N : i \notin I_k \text{ and } x \in S_i \,\}|\big)$
10     $x_k = \arg\max \{\, f_k(x) : x \in A_k \,\}$
11     $X_{k+1} = X_k \cup \{x_k\}$
12     $I_{k+1} = I_k \cup \{\, 1 \le i \le N : x_k \in S_i \,\}$
13     $A_{k+1} = \{\, x \in A_k : \text{it exists } i \notin I_{k+1} \text{ with } x \in S_i \,\}$
14     $k \leftarrow k + 1$
15 end
   Step 3 (patch):
16 $P = \mathrm{patchTermination}(q, X_k)$
17 return $P$
```

Algorithm 1 is an overview of TERMINATOR in pseudocode. The iteration counter for the greedy algorithm is k, the set $X_k \subseteq S$ is the current solution set, I_k is the set of trace indices that are already covered ($i \in I_k$ implies $X_k \cap S_i \neq \{\}$), and $A_k \subseteq S$ is the current set of acceptable basic blocks that may be added to the solution. In line 2, the initial acceptable set A_0 is set to the allowed subset of the trace union S. In lines 9 and 10, the basic block is selected that has the largest contribution to the objective function (8) by means of the auxiliary function $f_k\colon S \to \mathbb{R}^2$ defined by

$$f_k(x) = \big(g(x),\ |\{\, 1 \le i \le N : i \notin I_k \text{ and } x \in S_i \,\}|\big). \tag{10}$$

As before, the (argument) maximum in line 10 is to be understood with respect to the ordinary lexicographic order. Note that, in order to build the *smallest* solution set (among otherwise equally good candidates), it is desirable to select the basic block that appears in the *largest* number of traces that are not yet covered by the current solution X_k; this choice is why the algorithm is called greedy. Hence, it is correct to flip the sign in the second component of the tuple in line 9 compared to (8). In line 13, the basic blocks that only appear in traces already covered are removed from the acceptable set. Together with lines 5 and 7, this ensures that the algorithm terminates after at most N iterations.

D. Notes and technical implementation details

Once TERMINATOR has determined a solution of problem (9) as a set of termination blocks, the last step is to modify these basic blocks such that they terminate the running process when they are executed. This is done statically, that is, the basic blocks are rewritten in the binary files they are located in, which allows the modified program to run natively, i.e., without dynamic instrumentation.

Rewriting binary files is a delicate process that requires caution so as not to invalidate hard-coded offsets, for instance. This is why TERMINATOR does not *add* code to the selected basic block (making them larger), but instead *overwrites* them. The major complication, which is discussed in this section, is that the code that terminates the program can be too large to actually fit into a given basic block without partially overwriting adjacent or nested basic blocks. The following technical details are specific to 32-bit and 64-bit Windows 10 on x86, but can be ported to other systems.

1) Size requirement: When modifying binary code to introduce termination logic, the size of the termination instruction sequence is a critical consideration. The termination code must fit entirely within the selected basic block to avoid overwriting adjacent or nested code.

In our implementation, TERMINATOR supports single-byte (int3), 4-byte (fastfail), and 5 to 12-bytes (e.g. TerminateProcess) API-based process termination mechanisms.

The size requirement is implemented in TERMINATOR by means of the set $U \subseteq \mathbb{B}$ in (6). If $\operatorname{size}(x)$ denotes the size of a basic block $x \in \mathbb{B}$ in bytes, we ensure that $U \subseteq \mathbb{B}$ satisfies

$$U \subseteq \{\, x \in \mathbb{B} : \operatorname{size}(x) \geq s_{min} \,\}\,, \tag{11}$$

where s_{min} is the minimum required size based on the selected termination mechanism (1, 4, 5, or 12 bytes).

2) Nested basic blocks requirement: Unless one terminates the program with a single-byte instruction, it is not sufficient to choose a basic block that is large enough to hold the (multi-byte) termination instruction sequence. Basic blocks need not be disjoint in the machine code, and hence changing one basic block carelessly might also accidently affect another.

A common reason for basic blocks to overlap is that basic blocks may share a common suffix because of fall-through labels; for example, these appear in `if-then-else` and `switch` statements in a C-like language. We therefore only consider a basic block to be a termination block candidate if is large enough and does not overlap with other basic blocks. In other words, there is no control flow transfer from somewhere to the middle of the basic block. One way to do so is to only consider basic blocks that contain a large enough area without any basic block boundaries.

As before, the restriction to safe-to-patch basic blocks is implemented by setting the set U in (6) appropriately. In practice, TERMINATOR automatically ensures that

$$U \subseteq \{\, x \in \mathbb{B} : \operatorname{safe\,size}(x) \geq 12 \,\}\,, \tag{12}$$

which is almost (11), except that we define $\operatorname{safe\,size}(x)$ as the number of bytes from start of the basic block x to the start of the (spatial) next basic block; if there are no nested blocks inside x, we have $\operatorname{size}(x) = \operatorname{safe\,size}(x)$.

For our evaluation (Section IV-B), we terminated the PUTs using single-byte `int3` instructions, which eliminated size and nested basic blocks constraints as a practical concern. This allowed TERMINATOR to consider virtually all basic blocks as potential termination candidates.

3) Threads and modules: We obtain the program traces with a custom client for the dynamic instrumentation framework DynamoRIO [10]. The client has an option to only record basic blocks from prespecified binary modules (executables or dynamic link libraries), thus narrowing down the scope of the program. This allows to focus the analysis of TERMINATOR on the program itself, while ignoring code from third-party and operating system libraries. For reasons of efficiency, the client does not record the actual traces T_i, but only the premiere order mapping $o_i \colon S_i \to \mathbb{N}$.

The metrics based on trace coverage – i.e., o_i, r_i, g and G – are with respect to the recorded basic blocks. That means, if we only record basic blocks from the executable, say, `Acrobat.exe`, a trace coverage value of $r_i(x) = 0.99$ for a basic block x means that $99\,\%$ of the basic blocks from `Acrobat.exe` premiere before or at x in the i-th trace. This targeted approach is particularly beneficial when the primary focus of fuzz testing is the main executable rather than external libraries.

Globally, i.e., considering all basic blocks from all modules, the value $r_i(x)$ could be entirely different. For example, the value could be much smaller, if a large amount of code from a third-party GUI framework is executed for the first time after the main executable is already covered. This can increase the efficiency, when the primary target of the fuzz testing is the executable without external libraries. Conversely, if the parsing functionality resides in a separate module, TERMINATOR would still identify termination points in the main executable that occur after the processing of the module has completed since TERMINATOR looks for basic blocks that appear consistently late in all traces.

There are multiple ways to define the trace of a program and the order of the basic blocks if several threads execute in parallel. A simple option is to enforce a sequence by using only a single order map o_i for all threads with a shared mutex. This has the drawback of changing the program semantics potentially, but we consider this acceptable because dynamic instrumentation is hardly truly transparent in practice anyway.

IV. EVALUATION

We evaluated TERMINATOR across 15 Windows GUI applications spanning multiple categories:

- PDF viewers: Adobe Acrobat Reader 24.2.20933.0, SumatraPDF 3.1.2rel, MuPDF 1.7, XpdfReader 4.02, Slim PDF Reader 2.0.14, Foxit Reader 2024.2.2, STDUViewer 1.6.375, Kofax Power PDF Advanced 5.0.0.19
- Archive managers: WinRAR 7.1.0, ALZip 8.51.0
- Image viewers: FSViewer 7.8, Irfanview 4.67, BandiView 7.8.0.1
- Font/ISO tools: Birdfont 6.10.5, UltraISO 9.7.6

Our evaluation includes widely-used closed-source applications (Adobe Reader, Kofax Power PDF, WinRAR) and open-source programs (XpdfReader, SumatraPDF, MuPDF). The open-source programs allow validation against known vulnerabilities, while the closed-source applications demonstrate real-world applicability.

Our evaluation is guided by the following research questions:

RQ1 Fuzzing Efficiency: How does TERMINATOR compare to heuristic-based termination in terms of fuzzing throughput and speed of common code coverage acquisition?

RQ2 Effectiveness and Soundness: Does TERMINATOR's automated termination strategy allow for the discovery of vulnerabilities, or does it terminate prematurely? How well do TERMINATOR-patched binaries generalize to inputs not seen during training?

RQ3 Training and Corpus Requirements: What are the characteristics of an effective training corpus for TERMINATOR? How robust are TERMINATOR's generated termination solutions to variations in training data and program non-determinism?

RQ4 Determinism: Are the results reproducible across repeated executions of TERMINATOR?

A. Training Corpus Setup

For each file-type, we constructed representative training corpora. The PDF files, a key focus for several of our benchmarks, were drawn from two distinct supersets. The first superset, referred to as *min217*, consists of 217 PDF files obtained by a corpus minimization algorithm for fuzzing filtered from approximately 80 000 PDF files collected from diverse web sources. The second superset, referred to as *SafeDocs*, consists of 20 610 PDF files that have been crawled from various issue trackers of PDF-processing programs [1, 44]. The *completepdf* corpus, for PDF-related experiments, is the union of min217 and SafeDocs.

For the non-PDF PUTs, we assembled 10-20 files matching each program's supported file types (e.g., ZIP files for archive managers, PNG files for image viewers, TTF files for font tools, and ISO files for disk image utilities). Our results in RQ3 demonstrate that small, well-constructed training corpora of this size are sufficient for TERMINATOR to identify reliable termination points.

B. Fuzzing Setup

TERMINATOR is agnostic to the underlying fuzzing framework and can be adapted to work with any fuzzing architecture. While our primary evaluation uses a LibAFL-based setup, TERMINATOR's core principle of patched termination can be adapted, with considerations, for other Windows fuzzing architectures. For instance, with snapshot-based fuzzers like WinFuzz [40], directly patching the executable might interfere with the snapshotting mechanism because termination would prevent restoring the snapshot in-process. In such cases, TERMINATOR's analysis phase could still identify termination blocks, but instead of patching, the fuzzer's harness would monitor for execution of these blocks and then trigger a clean exit (e.g., sending a `WM_CLOSE` message to the main window) to allow the target function to return and maintain snapshot integrity. Fork-based fuzzers on Windows, such as Winnie [26], present different challenges. We observed in preliminary investigations that Winnie's process cloning method did not fully support WinAPI UI calls within cloned processes, which limits its direct applicability for fuzzing the main GUI-handling logic of applications where TERMINATOR would typically identify termination points.

We implemented our test environment using LibAFL [18] with breakpoint-based instrumentation as the fuzzing framework. This setup allows an easy integration of TERMINATOR's exit offset(s). In earlier exploratory phases of this research, we also successfully utilized WinAFL [19] with DynamoRIO-based instrumentation, which provided initial validation for TERMINATOR's fuzzing performance before transitioning to our more customized LibAFL setup for systematic evaluation.

C. Baseline Termination Strategies

Choosing an appropriate termination baseline for GUI applications is challenging. Fixed global timeouts are often suboptimal, as processing times vary significantly with input complexity, potentially leading to premature termination for complex files or inefficiency for simple ones. Monitoring system resources like CPU usage offer a more adaptive approach. While perfect idle detection is non-trivial (e.g., distinguishing temporary I/O waits from genuine post-processing idleness), CPU idle monitoring provides a pragmatic and widely applicable heuristic.

We implemented a CPU usage monitor (IDLE-EXIT) to determine appropriate timeout parameters for each application. We executed a calibration process that runs each target 100 times with random inputs while monitoring CPU usage for 15 seconds per run. The monitor counts consecutive intervals of 50ms where CPU usage stays below a threshold of 5%. The maximum observed streak across all runs determines the termination threshold for that specific application. Our calibration process revealed significant variation in idle behavior across applications, with maximum consecutive idle intervals ranging from 4 (MuPDF) to 190 (BandiView).

We have also considered an alternative heuristic to CPU monitoring: intercepting system calls related to waiting or message handling, such as 'NtWaitForSingleObject' or 'PeekMessage'. The intuition is that once the program begins waiting indefinitely, it has finished processing the file. We investigated this approach but found it to be less reliable and general than CPU monitoring for several reasons. First, GUI applications make hundreds of such calls during initialization and active processing, making it difficult to heuristically determine which specific call signifies the end of file parsing. Second, applications use a wide variety of APIs and custom event loops to wait for input. Our preliminary tests confirmed these challenges. We implemented hooks for 'GetMessage' (which internally calls 'NtWaitForSingleObject') and 'PeekMessage' and found their behavior to be inconsistent across applications. For BandiView, WinRAR, ALZip, and SlimPDFReader, 'PeekMessage' was used, but the message queue was already empty before the input file was even read. For STDUViewer, 'GetMessageW' was used, but it would block before file processing, as no messages were pending. Only for Adobe Reader did 'GetMessageW' appear to be a potential indicator of post-processing idleness. Given this lack of a consistent, reliable signal across diverse applications, we concluded that a syscall-based heuristic would require significant per-application tuning, undermining the goal of a general, automated solution. We thus selected the more broadly applicable CPU idle monitoring as our baseline.

D. Coverage Analysis Framework and Rationale

Our coverage analysis framework is specifically designed to evaluate fuzzing *efficiency* (RQ1) by comparing how quickly different termination strategies reach the same functional code paths. The choice of termination strategy inherently influences the scope of code executed. TERMINATOR is designed to terminate the PUT soon after essential input processing is complete, thereby deliberately avoiding extensive execution of subsequent code paths such as UI rendering, event loop idling, or lengthy cleanup routines. In contrast, IDLE-EXIT, which waits for CPU idleness, will naturally allow the PUT to execute more of these post-processing stages.

Let E represent the set of basic blocks corresponding to the core file-processing functionality and other essential logic that is relevant to the fuzzing campaign's goal of finding vulnerabilities in input handling. Let U represent auxiliary basic blocks primarily associated with UI rendering, prolonged idling, background tasks unrelated to the immediate input, or cleanup routines that are typically executed after the main input processing is finished. The IDLE-EXIT approach, by waiting for idleness, tends to cover code paths from both E and U, so its total coverage can be conceptualized as $O_{total} \supseteq E \cup U_{subset}$, where U_{subset} is the portion of U reached before termination. TERMINATOR, by design, aims to cover E efficiently and terminate before significant portions of U are executed, so its total coverage M_{total} will primarily consist of blocks from E.

This fundamental difference necessitates an evaluation methodology that focuses on meaningful coverage for vulnerability discovery. While IDLE-EXIT might achieve higher raw coverage numbers by executing more of U, this additional coverage provides limited value for finding bugs in the core file-processing logic. Therefore, our primary comparison of *efficiency* is based on the set of basic blocks discovered by *both* approaches.

Definition 1 (Common Coverage)**.** Let O_{total} represent the total set of unique basic blocks covered by the IDLE-EXIT approach over the entire fuzzing campaign, and M_{total} represent the total set of unique basic blocks covered by the TERMINATOR approach. The *common coverage*, denoted by C, is the intersection of these two sets:

$$C = O_{total} \cap M_{total}$$

This set C represents the essential functional code paths related to input processing that both termination strategies can reach. Our evaluation of fuzzing speed focuses on how quickly each approach accumulates coverage within C. This ensures we are comparing their efficiency in executing shared, core program logic, which is most relevant to the fuzzing objectives. TERMINATOR's goal is to execute these common paths more quickly. While IDLE-EXIT might explore additional blocks outside of C (e.g., extensive UI rendering or idling loops), these are precisely the paths TERMINATOR aims to avoid to improve throughput. As we show in Section V, this focused efficiency does not come at the cost of bug-finding capability.

The filtered cumulative coverage sets at any time t during a fuzzing run are then defined as:

$$F_O(t) = O_{cum}(t) \cap C \quad \text{and} \quad F_M(t) = M_{cum}(t) \cap C \quad (13)$$

where $O_{cum}(t)$ and $M_{cum}(t)$ represent the cumulative sets of unique basic blocks discovered by IDLE-EXIT and TERMINATOR respectively, from the beginning of the fuzzing run up to time t.

E. Statistical Methodology

To account for the inherently stochastic nature of fuzzing, we employed a robust statistical methodology:

1) For each target and approach combination, we conducted 3 separate 24-hour fuzzing runs (36 total fuzzing runs)
2) We compared each of the 3 TERMINATOR runs against each of the 3 baseline IDLE-EXIT runs (9 pairwise comparisons per target)
3) Speed advantage metrics were derived through bootstrap analysis with 10,000 resamples
4) 95% confidence intervals were calculated to assess statistical significance

This methodology provides significantly more robust results than single-run comparisons, properly accounting for variation between fuzzing runs.

F. Speed Advantage Calculation Methodology

The speed advantage factors we report represent a comprehensive measurement derived from runtime-based milestones, processed through several layers of statistical aggregation:

1) **Runtime milestone definition:** For each pairwise comparison, we define milestone times at fractions [0.1, 0.5, 0.625, 0.75, 0.875, 1.0] of the fuzzing run's total duration.
2) **Coverage percentage mapping:** For each runtime milestone t, we determine what percentage of common blocks the IDLE-EXIT approach has covered by that time.

TABLE II: Mean Speed Advantages (Ā), Throughput, and Unique Crashes encountered (Cr.) for TERMINATOR (T.) vs. IDLE-EXIT (I.) (24-hour runs, 3 repetitions per configuration).

Application	Ā	95% CI	T. exec/s	I. exec/s	T./I. Cr.
SumatraPDF	45.49	[14.38, 79.51]	1.52	0.05	0 / 0
UltraISO	36.18	[28.06, 44.28]	0.60	0.01	3 / 0
XpdfReader	12.81	[9.51, 16.78]	1.01	0.05	0 / 0
BandiView	12.28	[5.02, 21.39]	0.49	0.02	0 / 0
FSViewer	9.55	[7.59, 11.74]	0.85	0.20	2 / 4
SlimPDFReader	7.16	[3.98, 11.19]	0.16	0.03	2 / 2
ALZip	6.25	[4.12, 8.74]	1.08	0.18	1 / 0
Birdfont	5.79	[4.46, 7.01]	0.05	0.01	1 / 1
FoxitPDFReader	5.45	[4.46, 6.64]	0.10	0.03	2 / 1
STDUViewer	4.38	[3.38, 5.34]	0.78	0.07	6 / 2
Kofax PDF	2.26	[1.73, 2.93]	0.30	0.09	1 / 1
Irfanview	2.23	[1.93, 2.57]	1.36	0.41	0 / 0
Adobe Reader	2.10	[1.01, 3.52]	0.15	0.03	0 / 0
WinRAR	1.89	[1.07, 2.91]	0.72	0.38	0 / 0
MuPDF	1.61	[1.30, 1.93]	2.00	1.21	0 / 0

3) **Time-to-coverage comparison:** For each identified coverage percentage, we calculate how much faster TERMINATOR reached that same coverage level:

$$A(c_t) = \frac{t}{T_M(c_t)} \tag{14}$$

where c_t is the coverage percentage achieved by IDLE-EXIT at time t, and $T_M(c_t)$ is the time required for TERMINATOR to reach the same coverage percentage.

4) **Speed advantage aggregation:** For each pairwise comparison, we calculate the mean across all runtime-derived advantages:

$$\overline{A} = \frac{1}{|M|} \sum_{t \in M} A(c_t) \tag{15}$$

where M is the set of runtime milestone times.

5) **Statistical bootstrapping:** With 9 pairwise $\overline{A}$ values per application (from 3×3 comparisons), we apply bootstrap analysis with 10,000 resamples to derive the final mean advantage and 95% confidence intervals.

This runtime-based methodology ensures our reported speed advantages represent true performance improvements across the entire execution timeline, while accounting for the inherent variability in fuzzing outcomes. All experiments were performed on Windows Server 2025 VMs, using qemu [6] hosted on an AS-1125HS-TNR server with two AMD EPYC 9754 processors, allocating 4 cores and 16GB RAM per VM.

V. EXPERIMENTAL RESULTS

This section presents our findings, structured around the research questions outlined in Section IV.

A. RQ1: Fuzzing Efficiency

Table II presents the speed advantage (Ā) results for TERMINATOR compared to the IDLE-EXIT baseline across all 15 target applications, along with crash discovery comparison. Each result is based on 3 independent 24-hour fuzzing runs per application.

The results demonstrate that TERMINATOR consistently achieves substantial speed advantages in coverage acquisition across all tested applications, with mean advantages ranging from 1.6x to 45.49x.

The bootstrap analysis with multiple resamples per 24-hour fuzzing run provides strong confidence in our overall speed advantage results. The 95% confidence intervals in Table II show the robustness of our findings. Even at the lower bounds of these intervals, TERMINATOR demonstrates meaningful performance improvements across all applications. The narrower confidence intervals for applications like MuPDF [1.30, 1.93] and Kofax PDF [1.73, 2.93] indicate highly consistent speed advantages across different fuzzing runs. In contrast, wider intervals such as SumatraPDF [14.38, 79.51] and Adobe Reader [1.01, 3.52] reflect greater variability in performance gains, though all confidence intervals exclude 1.0, indicating statistically significant improvements.

To isolate the effectiveness of the exploration strategy from the raw throughput advantage, we also analyzed the code discovered within the first 1,000 executions of each fuzzing run. This analysis revealed that TERMINATOR not only covers common code faster but also discovers significantly more code overall. Aggregated across all 15 programs, TERMINATOR discovered 87,389 unique basic blocks, while IDLE-EXIT discovered only 62,558 in the same number of executions which indicates an increase of 39.7% for TERMINATOR. This suggests that by terminating executions precisely after parsing, TERMINATOR provides a sharper, more relevant feedback signal to the fuzzer's mutation engine, enabling it to explore deeper into complex parsing logic rather than getting distracted by post-processing UI code.

The consistency of these findings across application types and sizes provides strong evidence for the effectiveness and robustness of our coverage-guided termination approach.

B. RQ2: Effectiveness and Soundness

A key concern is whether optimizing for speed via automated termination leads to missing vulnerabilities. The crash discovery results in Table II demonstrate that TERMINATOR's termination logic does not compromise vulnerability detection effectiveness. TERMINATOR discovered more unique crashes than IDLE-EXIT in 4 out of 15 applications (UltraISO, STDUViewer, ALZip, and FoxitPDFReader), with just one case (FSViewer) where IDLE-EXIT found more crashes than TERMINATOR. This superior crash discovery performance shows that TERMINATOR's increased fuzzing throughput directly translates to improved vulnerability discovery.

Further bolstering this, for the PDF viewers XpdfReader, MuPDF, and SumatraPDF, TERMINATOR-modified binaries were tested against several known CVEs (CVE-2016-6525, CVE-2017-15587 for MuPDF/SumatraPDF; CVE-2019-16088, CVE-2019-17064 for XpdfReader). In all configurations, the TERMINATOR-modified programs successfully triggered these known vulnerabilities. This provides direct evidence that TERMINATOR's termination points do not typically cut off vulnerable code paths. The guaranteed trace coverage $G(X)$

TABLE III: Minimum effective training corpora for different PDF viewers. "Start" indicates the initial corpus, "Blocks" shows the number of basic blocks in the termination solution, and "Files" shows the minimum number of files needed for an effective solution.

#	Program	Start Corpus	Solution Blocks	Min. Requ. Files
1	SumatraPDF	min217+12	11	12
2	MuPDF	min217+12	9	18
3	XpdfReader	min217+12	10	8
4	XpdfReader	min217-1+12	4	7

for solutions generated by TERMINATOR was consistently high (e.g., >99.9% for MuPDF/XpdfReader in specific tests), indicating that very little of the originally traced code is missed.

Regarding soundness and generalization, TERMINATOR's approach demonstrates good performance. When trained on a relatively small and diverse corpus (200 files for PDF viewers), the resulting patched binaries successfully terminated for over 98-99% of inputs from a much larger, unseen corpus of over 20,000 files (*SafeDocs*). The few non-terminating cases were typically due to program states not represented in the training set, such as password prompts for encrypted files or unique error handling paths. Augmenting the training corpus with a small number of such examples (e.g., 12 diverse files) further improved the success rate to over 99.9%, with remaining non-terminations usually corresponding to actual hangs or crashes (the very targets of fuzzing).

C. RQ3: Training and Corpus Requirements

The effectiveness of TERMINATOR depends on the quality and composition of its training corpus. We conducted detailed experiments to determine: (1) the minimum corpus size required for effective termination points, (2) the impact of corpus composition on solution quality, and (3) guidelines for optimal training corpus creation.

We began with our standard training corpus (min217+12) containing 217 minimized PDF files plus 12 specially crafted files for error cases. To identify the minimum required corpus size, we employed a reduction technique: files were iteratively removed in order of their trace coverage (highest first), stopping when further removal would change the termination solution. This process revealed that surprisingly few files (between 8-18) were necessary to generate effective termination solutions (Table III, rows 1-3).

We then conducted qualitative analysis by manually examining each file in the minimal corpus with the original, unmodified programs. For MuPDF's minimum corpus (18 files), we found that 12 files displayed normally without errors, one file was encrypted (triggering password prompts), and seven files generated various warnings or error messages. This diversity suggests an effective training corpus needs to cover both normal operation and various error-handling paths.

Further experiments demonstrate that TERMINATOR can produce correct termination points with as few as 2 files in the training corpus. This efficiency stems from TERMINATOR's approach of identifying basic blocks that appear consistently late in all traces. Larger training sets typically yield more robust solutions for diverse input formats.

A critical discovery emerged when analyzing XpdfReader's solution. One of the eight files in its minimum corpus caused a segmentation fault in a destructor during error handling. When this file was included in training, TERMINATOR identified termination points that would cut off execution before this fault occurred (prematurely terminating execution) — evidenced by the relatively low trace coverage (87.0% and 76.8%) in the resulting solution.

To verify this finding, we removed this problematic file from the corpus and re-ran our analysis ("min217-1+12"). The resulting solution ① required fewer files (7 versus 8), ② used fewer termination blocks (4 versus 10), ③ Achieved significantly higher trace coverage (99.6% versus 82% average), ④ Properly exposed both known vulnerabilities in testing, ⑤ maintained the same success rate on the complete corpus.

This experiment highlights that files that cause crashes or hangs should be excluded from the training corpus. Including such files can cause TERMINATOR to learn termination points that occur before crash-triggering code paths are fully explored, potentially hiding the very vulnerabilities that fuzzing aims to discover. The optimal training corpus should contain files that exercise program functionality without causing crashes.

Based on these findings, our implementation of TERMINATOR automatically excludes any inputs that cause crashes or hangs during trace collection. This automatic filtering ensures that the chosen termination points will be consistently sound: They will always occur after complete input processing but before waiting for user interaction. By excluding crashing inputs from training, TERMINATOR structurally prevents premature termination, guaranteeing that vulnerable code paths remain fully exposed during fuzzing.

To address cases where poor training data quality leads to TERMINATOR failing to terminate for unseen inputs, our LibAFL based fuzzer contains a fallback mechanism that reverts to CPU idle monitoring when TERMINATOR's coverage-guided termination does not trigger within a reasonable timeout. This hybrid approach ensures robust operation and prevents the fuzzer from hanging.

D. RQ4: Determinism and Robustness

Non-determinism in the PUT, i.e., repeated executions of the PUT with identical inputs resulting in different traces, may load to TERMINATOR finding different solutions when run multiple times. Here, we investigate the extent to which the non-determinism of PUT affects TERMINATOR.

We created a training corpus of 10 random input files from the `completepdf` corpus and ran TERMINATOR 100 times for the three PDF programs (XpdfReader, MuPDF and SumatraPDF). The results for XpdfReader are shown in Table IV. TERMINATOR found five different solutions (rows 1 to 5), of which three were only found once. The solutions are very similar as the *Symbol* column proves: they all use a selection of the same four basic blocks. The minimum trace

TABLE IV: TERMINATOR solutions for 100 repeated executions. The column n displays the prevalence of the solution X. The *Symbol* column shows the function name and (file) line number of the basic blocks as a reference.

#	n	μ	σ	Symbol
1	78	99.97	0.02	`XpdfV::statusIndicatorStop#3636`
2	19	99.97	0.02	`XpdfV::statusIndicatorStop#3636` `TileCompositor::getBitmap#127`
3	1	99.74	0.03	`TileCompositor::blit#323` `TileCompositor::getBitmap#127`
4	1	99.97	0.02	`XpdfV::statusIndicatorStop#3636` `TileCompositor::blit#332` `TileCompositor::getBitmap#127`
5	1	99.74	0.03	`TileCompositor::blit#323` `TileCompositor::blit#332` `TileCompositor::getBitmap#127`
6	100	99.97	0.02	`XpdfV::statusIndicatorStop#3636`

coverage $G(X)$ of a solution X is with respect to a given set of traces; for each solution, we have computed that value for all 100 sets of traces. The mean values μ of the solutions are similar, and the standard deviation σ is less than $0.04\,\%$. Therefore, the solutions are exchangable in practice.

The results for SumatraPDF and MuPDF were similar in that there was no significant variance in the solutions. For the three test programs, our data does not indicate any issues due to non-determinism. In general, non-determinism can be addressed by the size of the training corpus and by collecting several traces for each input file instead of just a single one.

VI. Limitations and Threats to Validity

A. Limitations

While TERMINATOR significantly improves the fuzzing of GUI applications, several limitations should be acknowledged:

1) Multi-process Applications: TERMINATOR currently focuses on the main process for trace collection and termination. For applications with multi-process architectures (e.g., web browsers), additional engineering effort is required to identify termination points across processes.

2) Multi-threaded Applications: TERMINATOR currently enforces sequential traces with a shared mutex. This approach simplifies trace collection but introduces potential non-determinism. Multiple runs of the same inputs during training minimizes possible issues. Our evaluation shows this limitation does not impact effectiveness.

3) Training Corpus Dependency: The quality of termination points depends on training corpus diversity. If the training set lacks examples of certain program behaviors (e.g., password prompts, specific error conditions), TERMINATOR may select suboptimal termination points for those cases that will not terminate when encountering unusual program behaviors. While our experiments show small diverse corpora (10-20 files) suffice, careful corpus curation remains important.

B. Threats to Validity

Application Selection: We evaluated 15 Windows programs across different categories. While diverse, this set may not represent all GUI application architectures. Applications with unusual input processing patterns might show different results.

Platform Specificity: Our implementation targets Windows x86/x64 binaries. The approach should generalize to other platforms, but implementation details (e.g., process termination methods, trace collection) would differ.

File Format Coverage: Our evaluation emphasizes PDF viewers (8 of 15 applications) due to their security relevance and CVE availability. Other file formats might exhibit different characteristics affecting TERMINATOR's effectiveness.

VII. Conclusion

We presented TERMINATOR, a novel approach for enabling efficient fuzzing of GUI-based file-processing applications through coverage-guided automatic termination. Our method eliminates the need for manual harness development or unreliable heuristics by analyzing code coverage patterns and automatically modifying binaries to terminate at optimal points. The evaluation demonstrates that TERMINATOR significantly improves fuzzing efficiency compared to traditional CPU-based heuristics, achieving speedups of 1.6x to 45.49x across 15 diverse applications while maintaining the ability to reach vulnerable code paths. Crucially, we verified TERMINATOR's reliability by successfully reproducing known CVEs in multiple PDF viewers.

Future work could explore dynamic taint analysis for more precise termination decisions and investigate deeper integration with other fuzzing architectures. Combining TERMINATOR's automated termination signal with the high-throughput execution of snapshot-based fuzzers or the deterministic replay capabilities of environment fuzzers presents a promising direction. Consequently, TERMINATOR opens new possibilities for testing complex GUI applications that have traditionally resisted automated analysis.

A. Artifact Availability

To facilitate reproducibility and enable further research release TERMINATOR as open-source software: https://github.com/Fraunhofer-SIT/ASE2025-Terminator.

Acknowledgment

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Tim Allison et al. *Building a Wide Reach Corpus*. Research Report for the IEEE Security & Privacy LangSec Workshop. NASA Jet Propulsion Laboratory, 2020, p. 9.

[2] Yoav Alon and Netanel Ben-Simon. Dec. 12, 2018. URL: https://research.checkpoint.com/2018/50-adobe-cves-in-50-days/ (visited on 08/28/2020).

[3] Cornelius Aschermann et al. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 1597–1612.

[4] Cornelius Aschermann et al. "NAUTILUS: Fishing for Deep Bugs with Grammars". In: *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019.

[5] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019.

[6] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *FREENIX Track: 2005 USENIX Annual Technical Conference*. Jan. 2005, pp. 41–46.

[7] Tim Blazytko et al. "GRIMOIRE: Synthesizing Structure while Fuzzing". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1985–2002.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 24, 2016, pp. 1032–1043.

[9] Marcel Böhme et al. "Directed Greybox Fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. Dallas, Texas, USA: ACM Press, 2017, pp. 2329–2344.

[10] Derek Bruening, Qin Zhao, and Saman Amarasinghe. "Transparent Dynamic Instrumentation". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. event-place: London, England, UK. New York, NY, USA: Association for Computing Machinery, 2012, pp. 133–144.

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

[12] Hongxu Chen et al. "Hawkeye: Towards a Desired Directed Grey-box Fuzzer". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Jan. 15, 2018, pp. 2095–2108.

[13] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA: IEEE, May 2018, pp. 711–725.

[14] Peng Chen, Jianzhong Liu, and Hao Chen. "Matryoshka: Fuzzing Deeply Nested Branches". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, Nov. 6, 2019, pp. 499–513.

[15] Yaohui Chen et al. "SAVIOR: Towards Bug-Driven Hybrid Testing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 1580–1596.

[16] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Guiding dynamic symbolic execution toward unverified program executions". In: *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. Austin, Texas: ACM Press, 2016, pp. 144–155.

[17] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[18] Andrea Fioraldi et al. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*. CCS '22. Los Angeles, U.S.A.: ACM, Nov. 2022.

[19] Ivan Fratric. *WinAFL*. 2020. URL: https://github.com/googleprojectzero/winafl.

[20] Gerald Gamrath et al. *The SCIP Optimization Suite 7.0*. Technical Report. Optimization Online, Mar. 2020.

[21] Shuitao Gan et al. "GREYONE: Data Flow Sensitive Fuzzing". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2577–2594.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing". In: *Queue* 10.1 (Jan. 2012). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 20–27.

[23] Google. *OSS-Fuzz*. 2020. URL: https://github.com/google/oss-fuzz/blob/8e5f1444661edd32964e7f156f9d5e3ee724c69b/README.md (visited on 08/25/2020).

[24] Peter Gutmann. "Fuzzing Code with AFL". In: *;login: Summer 2016* Volume 41 (Number 2 2016), pp. 11–14.

[25] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines". In: *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019.

[26] Jinho Jung et al. "WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning". In: Jan. 2021.

[27] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1.1 (Dec. 2018), p. 6.

[28] Hongliang Liang et al. "Fuzzing: State of the Art". In: *IEEE Transactions on Reliability* 67.3 (Sept. 2018), pp. 1199–1218.

[29] *libFuzzer – a library for coverage-guided fuzz testing*. URL: https://llvm.org/docs/LibFuzzer.html (visited on 08/28/2020).

[30] Paul Dan Marinescu and Cristian Cadar. "KATCH: high-coverage testing of software patches". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. Saint Petersburg, Russia: ACM Press, 2013, p. 235.

[31] Björn Mathis et al. "Parser-directed fuzzing". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*. Phoenix, AZ, USA: ACM Press, 2019, pp. 548–560.

[32] Richard McNally, Ken Yiu, and Duncan Grove. *Fuzzing: The State of the Art*. Technical report DSTO–TN–1043. Edinburgh, Australia: DSTO Defence Science and Technology Organisation, Feb. 2012, p. 55.

[33] Ruijie Meng, Gregory J. Duck, and Abhik Roychoudhury. "Program Environment Fuzzing". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, pp. 720–734.

[34] Xiaozhu Meng and Barton P. Miller. "Binary code is not easy". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. Saarbrücken, Germany: ACM Press, 2016, pp. 24–35.

[35] Barton P. Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (Dec. 1, 1990), pp. 32–44.

[36] Rohan Padhye et al. "Validity Fuzzing and Parametric Generators for Effective Random Testing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Montreal, QC, Canada: IEEE, May 2019, pp. 266–267.

[37] Sebastian Poeplau and Aurélien Francillon. "Symbolic execution with SymCC: Don't interpret, compile!" In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198.

[38] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing". In: *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017.

[39] StatCounter. *Desktop Operating System Market Share Worldwide*. Accessed: 2024-10-30. 2024. URL: https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-202402-202402-bar.

[40] Leo Stone et al. "No Linux, No Problem: Fast and Correct Windows Binary Fuzzing via Target-embedded Snapshotting". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4913–4929.

[41] Peach Tech. *Peach Fuzzer*. 2020. URL: https://www.peach.tech/products/peach-fuzzer/.

[42] Van-Thuan Pham et al. "Smart Greybox Fuzzing". In: *IEEE Transactions on Software Engineering* (2019).

[43] Yanhao Wang et al. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization". In: *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020.

[44] Peter Wyatt. *A new stressful PDF corpus*. Sept. 10, 2020. URL: https://www.pdfa.org/a-new-stressful-pdf-corpus/ (visited on 10/26/2020).

[45] Insu Yun et al. "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761.

[46] Michał Zalewski. *american fuzzy lop*. 2020. URL: http://lcamtuf.coredump.cx/afl/.

[47] Lei Zhao et al. "Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing". In: *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019.