Docovery: Toward Generic Automatic Document Recovery

Tomasz Kuchta, Cristian Cadar
Imperial College London
Department of Computing
180 Queen's Gate, London, UK
{t.kuchta,c.cadar}@imperial.ac.uk

Miguel Castro, Manuel Costa
Microsoft Research
21 Station Road
Cambridge, UK
{mcastro,manuelc}@microsoft.com

ABSTRACT

Application crashes and errors that occur while loading a document are one of the most visible defects of consumer software. While documents become corrupted in various ways—from storage media failures to incompatibility across applications to malicious modifications—the underlying reason they fail to load in a certain application is that their contents cause the application logic to exercise an uncommon execution path which the software was not designed to handle, or which was not properly tested.

We present DOCOVERY, a novel document recovery technique based on symbolic execution that makes it possible to fix broken documents without any prior knowledge of the file format. Starting from the code path executed when opening a broken document, DOCOVERY explores alternative paths that avoid the error, and makes small changes to the document in order to force the application to follow one of these alternative paths.

We implemented our approach in a prototype tool based on the symbolic execution engine KLEE. We present a preliminary case study, which shows that DOCOVERY can successfully recover broken documents processed by several popular applications such as the e-mail client `pine`, the pagination tool `pr` and the binary file utilities `dwarfdump` and `readelf`.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering;
E.5 [**Files**]: Backup/recovery

General Terms

Reliability

Keywords

Data recovery; symbolic execution; program analysis

1. INTRODUCTION

The inability to load text documents, e-mails, photos or music files in a desired application is one of the most user-visible defects of consumer software. This type of error can have diverse causes: files can be inadvertently corrupted by storage media failures, faulty network transfers or power outages; documents created by one application cannot be opened in another purportedly compatible application; code may have bugs that are triggered by specific file contents; or documents can be intentionally altered by attackers to exploit security vulnerabilities. One example from the last category is a known vulnerability in the `pine` e-mail client in which an e-mail with a maliciously crafted *From* field causes older versions of `pine` to abort execution while loading it, preventing users from accessing e-mails.

Regardless of how the problematic document was generated, the underlying reason it crashes or fails to load in a certain application is that its contents cause the application logic to exercise an uncommon execution path which the software was not designed to handle, or which was not properly tested.

Sometimes, the file can be repaired by fixing its broken structure; however, this requires a special recovery component that contains a parser for the given file format. While some popular document readers come with such customised recovery components, this is the exception rather than the rule.

In this paper, we introduce DOCOVERY, a novel document recovery technique which *does not require any prior knowledge of the underlying file format*. DOCOVERY takes as input a broken document and an application, and uses symbolic execution to fix the document to achieve two key goals: (1) the recovered document does not cause a failure; and (2) it is similar to the broken document. While creating a document that does not make the program crash is a prerequisite for a successful recovery, DOCOVERY goes beyond this. It attempts to create a document that avoids the crash while retaining as much of the content of the original document as possible.

We have implemented our approach in a prototype tool, which we have successfully used to recover documents processed by several medium-sized applications such as `pine`, `pr`, and `readelf`.

The rest of this paper is organised as follows. In §2 we define the document recovery problem and give a high-level overview of our technique. We present the various stages and challenges of the technique in §3, discuss implementation details in §4 and evaluate our prototype in §5. We discuss the main limitations of our approach in §6, present related work in §7 and conclude in §8.

2. OVERVIEW

Before describing our document recovery technique in detail, we further define the problem and introduce some terminology. For the purpose of this paper, the term *document* is used to refer to an

```
1  // file contents: "55"
2  FILE* f = fopen("f.txt", "r");
3  char x = fgetc(f);
4  char y = fgetc(f);
5  if (x >= '5') {
6    if (y >= '5') {
7      // CRASH!
8    } else {
9      // OK!
10   }
11 } else {
12   // OK!
13 }
```

Figure 1: A toy application.

input file processed by a certain program. Documents can be text or binary files.

We consider a document to be *broken* if it cannot be loaded successfully by an application. Note that our definition is relative to a given application—that is, a document might be broken in one application, but not necessarily in another. For example, a file may crash an older version of some document processing library but be handled correctly by newer versions of the same library or by other parsers of the given document format. In this paper, we often refer to the broken document as the *original document*.

We consider that a document is successfully loaded if the application (1) does not crash and (2) does not terminate with an error message or an error exit code. If available, more sophisticated oracles can be used, and in many cases these can be easily added by users or developers.

The aim of the document recovery process is to change the document so that it is successfully loaded by the application, while retaining a high level of similarity between the original document and the recovered document. The degree of similarity between two documents ultimately depends on the specific file type. In the absence of a high-level document specification, we use a *byte-level similarity metric*, which we discuss in §3.6.

In our approach, we attempt to recover a broken document without any prior knowledge of the file format, *i.e.*, without any document specification. Instead, we use the application code itself to understand how the program reads the document and what input bytes cause the loading error. For instance, if the application crashes after checking unsuccessfully that the sum of the first two bytes in the document is 15, the first two bytes should be altered such that they do sum to 15. DOCOVERY treats the document as an array $D[N]$ of N bytes, which can be formalised as a sequence of length N over an alphabet $A = \{0, 1, ..., 255\}$.

Our approach makes use of *symbolic execution* [6, 15], a popular program analysis technique that can precisely analyse and explore program execution paths. On each explored path, symbolic execution can gather exact mathematical constraints characterising all inputs which take the program along that path. As an example, consider the program in Figure 1. The program opens a text file, reads the first two characters and stores them in variables *x* and *y*. If both characters are greater than or equal to *'5'*, the program crashes. When symbolic execution follows this path, at each branch point it gathers the constraints on the input that take the program along this path: $x \geq$ *'5'* on line 5 and $y \geq$ *'5'* on line 6. As a result, the crashing path is precisely characterised by the conjunction of the constraints collected at each branch point: $(x \geq \text{'5'}) \wedge (y \geq \text{'5'})$.

Table 1: Paths explored in the toy application of Figure 1, with corresponding constraints and possible recovery candidates. The first row represents the original broken document.

Path (lines)	Constraints	File
...5, 6, 7	$x \geq \text{'5'} \wedge y \geq \text{'5'}$	'55'
...5, 11, 12	$x < \text{'5'}$	'05'
...5, 8, 9	$x \geq \text{'5'} \wedge y < \text{'5'}$	'50'

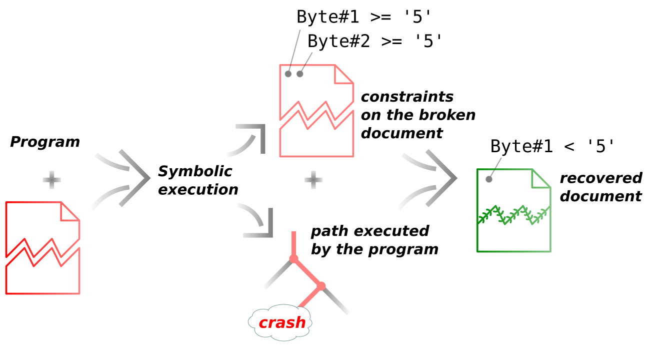

Figure 2: A high-level overview of DOCOVERY.

In symbolic execution, the program is executed on *symbolic* rather than concrete input and variables are represented as expressions over the symbolic input. On each executed path, symbolic execution maintains a *path condition (PC)* formula that characterises all the inputs that follow that path. New constraints are added to the PC when the execution reaches a branch point for which both sides are feasible under the current PC: at that point, execution is forked, following each side of the branch separately and adding the constraint that the branch condition is true to the PC of the *then* side, and that it is false to the PC of the *else* side.

In order to decide whether both sides of a branch are feasible, a *constraint solver* [11] is used to solve the logical formula in the current PC for satisfiability. A formula is satisfiable if there exists an assignment of concrete values to variables that makes the formula *true*. These concrete values satisfying the PC are called a *satisfying assignment* and are generated by the constraint solver. They represent an actual input that will drive execution along the same path on which that PC was collected. In our example, the formula $(x \geq \text{'5'}) \wedge (y \geq \text{'5'})$ is satisfiable, because there exists a satisfying assignment, *e.g.*, $x = 5, y = 5$.

In DOCOVERY, whose high-level overview is illustrated in Figure 2, we start by executing the program on the broken document, gathering constraints on the side, as in the *concolic* variant of symbolic execution [14, 26]. In other words, we treat program input as symbolic and execute the program symbolically, but at each branch point, instead of forking and exploring both sides of the branch, we use the concrete input values to decide which side of the branch to take. By using the broken document as a program input that guides the execution, we reach the point of program crash and collect a PC that corresponds to the execution path exercised by the broken document. Assuming the contents of the original document are *'55'*, then the collected PC will be $(x \geq \text{'5'}) \wedge (y \geq \text{'5'})$, as discussed above. This condition encodes *all* inputs that follow that program path and cause the application to crash. The relationship between concrete input values and the constraints in the PC is noteworthy: the concrete values determine the execution path, while the constraints describe all the input values that are valid on this path.

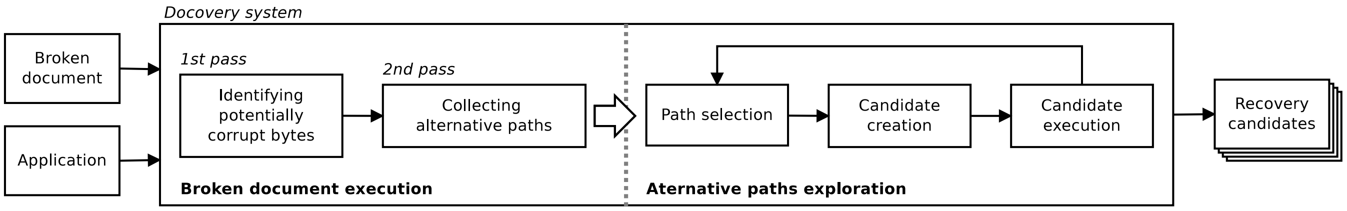

Figure 3: Main stages of the document recovery process.

In order to avoid the crash, our approach explores *alternative execution paths* around the one executed by the broken document by systematically negating one of the constraints in the PC, say the k^{th} constraint, and dropping the remaining constraints, from the $(k+1)^{st}$ to the last one. This simulates an execution in which the program follows the path executed with the original document until it reaches the k^{th} conditional statement, where it takes the other side of the branch. In our example, there are two choices: the first one is to negate the first constraint and drop the second, obtaining the PC $\neg(x \geq \text{'5'})$, corresponding to the path following the lines $2, 3, 4, 5, 11, 12$; and the second one is to negate the second constraint, obtaining the PC $(x \geq \text{'5'}) \wedge \neg(y \geq \text{'5'})$, corresponding to the path following the lines $2, 3, 4, 5, 6, 8, 9$. On each path, our technique takes the modified PC and solves it to generate a document similar to the original one. For example, the first PC could be solved to obtain document *'05'*, which differs only in the first character, while the second PC could be solved to obtain document *'50'*, which differs only in the second character. Both recovered documents avoid the crash, and thus represent valid *recovery candidates*, which can be presented to the user. Table 1 shows the PCs and possible recovery candidates associated with each explored path.

While our high-level approach is conceptually simple, there are several challenges that need to be addressed when it is scaled to real applications, which we discuss in the next section.

3. DOCOVERY

Figure 3 shows the main stages of our document recovery technique DOCOVERY. The inputs to the recovery process are the broken document and the source code of the application that fails to load the document.[1] The process consists of two main stages: broken document execution and alternative paths exploration. The output of DOCOVERY is a set of recovery candidates.

During the initial *broken document execution* stage, the application is executed twice. In the first execution pass, DOCOVERY identifies the bytes in the document that are potentially responsible for the failure (§3.1). On the second execution pass, the bytes identified in the first pass are marked as symbolic and a limited number of alternative paths, starting *N* branch points before the failure, are collected along with the associated PCs (§3.2). Since real applications and documents can provide a huge number of possible alternative execution paths, it is important to limit and prioritise the paths that will be processed.

In the second stage, the alternative execution paths collected during the initial stage are explored in a search for a correct (*i.e.*, not ending in a failure) program execution. The exploration process is iterative and consists of three steps. First, a path to be explored is selected (*Path selection*, §3.3). Second, a candidate document that follows the selected path is generated (*Candidate creation*, §3.4). Third, the candidate document is loaded in the original application in order to check whether it loads correctly (*Candidate validation*, §3.5). The candidates that are successfully verified in the third step are added to the pool of *recovery candidates*.

[1]Conceptually, the technique can run directly on binaries, but in our prototype we require source code. An example of a system implementing similar techniques for binaries is presented in [9].

Table 2: Time needed to get the first recovery candidate when the whole document is symbolic ('Whole') and when only the potentially corrupt bytes are symbolic ('Partial').

Benchmark	Whole	Partial
`pr`	timeout (3600s)	5.1s
`pine`	timeout (3600s)	338.9s
`dwarfdump`	timeout (3600s)	2.8s
`readelf`	14.8s	< 1s

3.1 Identifying potentially corrupt bytes

In theory, one could simply treat the whole broken document as symbolic and explore all feasible alternative paths. However, this could lead DOCOVERY to generate a large number of complex constraints, and to explore many alternative paths.

DOCOVERY aims to reduce the amount of symbolic data by treating as symbolic only those bytes which, when appropriately changed, are likely to result in a recovery candidate. In order to identify such potentially corrupt bytes, DOCOVERY uses a form of *dynamic taint tracking* [7, 25], a technique which tracks the flow of information from a set of sources—in our case the bytes in the document—to a set of sinks—in our case the computation where the failure manifests itself. The tracking is performed by associating a unique token (*'taint'*) with each byte in the input file, and then propagating these taints whenever an instruction is executed—*e.g.*, the instruction `x=file[0]+file[1]` would propagate the taints of `file[0]` and `file[1]` to variable `x`. The final result of this analysis is a set of document bytes whose values are likely to be involved in the failure.

Our analysis is both unsound (*i.e.*, it may miss relevant bytes) and imprecise (*i.e.*, it may include irrelevant ones). In general, we have opted for higher precision, that is, we tried to minimise the number of irrelevant bytes included; this limits the search space for recovery candidates, because fewer bytes are made symbolic and thus fewer bytes can be changed to create recovery candidates. In our evaluation (§5), we have never encountered a situation in which too few bytes were selected to allow recovery, but if this happens, it is possible to revert to a conservative taint analysis or simply include all the bytes in the document.

The design decisions that we have taken for our taint tracking implementation are as follows, all motivated by the goal of minimising the number of bytes selected by the analysis:

- **Byte-level precision**. Our taint tracking mechanism operates at the level of individual bytes, *i.e.*, for every tracked byte we store the information about a set of document bytes that might have influenced the value of that byte.

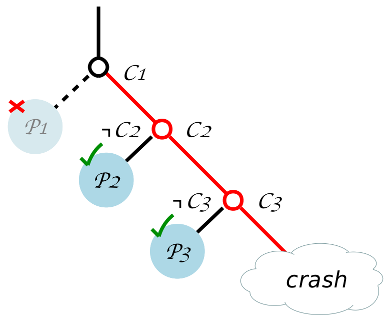

Figure 4: Illustration of alternative execution paths. The rightmost path is the one followed by the broken document, and c_1, c_2 and c_3 are the constraints collected on this path. Paths p_1, p_2 and p_3 are alternative execution paths: p_2 and p_3 are feasible, while p_1 is infeasible.

- **No control-flow dependencies**. We only track data-flow dependencies, *i.e.*, we do not propagate taints that are associated with control-flow branch points.

- **No address tainting.** When an address (pointer) is computed based on the data in the input document, we do not propagate taints from the address to the target of a read/write memory operation.

To illustrate the computational overhead associated with treating the entire document as symbolic, we performed experiments using the smallest file in each of our benchmarks (which are described in §5). We measured the time taken to generate the first recovery candidate when the whole document is treated as symbolic, and when only the bytes identified by our taint tracking analysis are treated as symbolic. In both cases, we set a timeout of one hour and collected at most 500 alternative paths in the manner described in §3.2 and §3.3.

Table 2 shows that taint analysis significantly reduces the time needed to generate the first recovery candidate. In particular, when treating the whole document as symbolic in `pr`, `pine` and `dwarfdump`, DOCOVERY cannot generate any recovery candidates within one hour, whereas when treating only the identified bytes as symbolic, DOCOVERY generates a recovery candidate in 5.1s, 338.9s and 2.8s, respectively.

3.2 Lazily collecting alternative paths

After the potentially corrupt bytes are identified, the next step involves treating these bytes as symbolic and running the application on the broken document in order to collect *alternative execution paths*, *i.e.*, those paths where execution could diverge from the one followed by the broken document if the identified bytes were changed. The process is illustrated graphically in Figure 4.

While each branch point involving symbolic data could potentially provide an alternative path, not all such paths are feasible. For example, if the program first follows a branch on which $x > 10$ and afterwards encounters the branch point *if* $x > 5$, then the alternative path ($x \leq 5$) is not feasible at this point, as x is already constrained to be > 10 (and thus > 5).

Determining whether an alternative path is feasible requires a call to the constraint solver, which is expensive. With the large number of branch points encountered on real execution paths, this would likely exhaust the entire time budget at this stage. As a result, our approach is to collect all paths diverging from the path followed

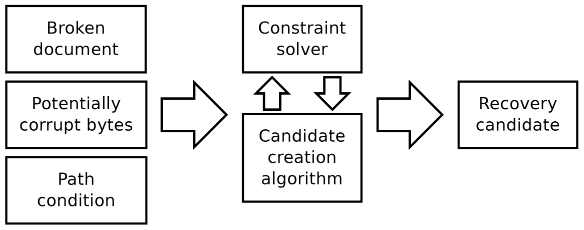

Figure 5: Overview of the recovery candidate creation process.

by the broken document—both feasible and infeasible—and *lazily* verify their feasibility only when they are selected for execution.

The drawback of this lazy approach is that we potentially need to store a large number of paths and memory consumption can become an issue. In order to tackle this problem, our strategy is to store only the last *N* alternative paths for further processing. The intuition behind this strategy is that alternative paths which are closer to the fault are more likely to result in candidate documents that are similar to the original document, because their PCs will share a larger prefix with the PC of the original execution (which is already satisfied by the bytes in the original document).

To illustrate the need for lazily collecting paths, let's consider two experiments on the `readelf` debug information display utility, one of the benchmarks we explore in §5. For the first experiment, we selected a small 54KB file which we marked as symbolic in its entirety. When the constraint solver was used to check alternative path feasibility at branch points, the first candidate was generated after 1,267s, compared with only 14.8s when lazy path collection was used.

To show the need to limit the number of collected paths, we ran another experiment, this time using a much larger 1.5MB file, in which we only marked as symbolic the bytes identified by taint tracking, and collected paths in a lazy manner. When only the last 500 alternative paths were collected, the time needed to create the first recovery candidate was approximately 46s and the memory consumption was below 172MB. However, when all the alternative paths were collected, it took approximately 242s to create the first candidate and the memory consumption reached 7.3GB. This large memory footprint is due to the large number of alternative paths stored, which was over 67,000 in this case.

3.3 Path selection

Our approach to path selection is to choose paths in decreasing depth: we start from the deepest path and after that one is explored we take the next deepest and so on, until all *N* collected alternative paths are exhausted. In the example from Figure 4, path p_3 will be selected first, then path p_2. Similar to the argument presented in §3.2, the rationale behind this path selection strategy is to first choose paths that are closer to the bug, *i.e.*, those which share a longer prefix of the PC with the original execution path. Intuitively, since the bytes in the original document already satisfy the conditions on the shared prefix, this strategy should reduce the number of bytes that need to be changed to make execution follow the alternative path. Path p_3 shares constraints c_1 and c_2 with the original execution path, whereas path p_2 has only constraint c_1 in common.

3.4 Creating recovery candidates

An essential part of DOCOVERY is the creation of recovery candidates that take the program along an alternative execution path and retain a high level of similarity to the broken document. A diagram presenting DOCOVERY's candidate creation process is depicted in Figure 5. There are several inputs to the process:

- **The broken document.** All the bytes that do not need to be changed in the candidate file in order to follow the alternative path are left unchanged.
- **The set of potentially corrupt bytes**, as identified by the taint tracking algorithm presented in §3.1. These are the bytes that are made symbolic in order to explore alternative paths.
- **The path condition (PC) associated with a given alternative path**, which is fed to a constraint solver to select new values for the symbolic bytes that make execution follow the alternative path.

The output of the recovery candidate creation process is a recovery candidate document that (1) obeys the constraints in the PC of the alternative path and (2) is similar to the broken document. Another way to see this is that we want to make the modified document satisfy the PC of the alternative path while changing as few bytes as possible.

There are two possible strategies that can be employed to create recovery candidates: a satisfying assignment-based approach, which is simple, but also potentially changes more bytes than necessary, and a precise algorithm, which changes fewer bytes but may require more calls to the constraint solver that can be expensive. Below, we present each of these strategies.

3.4.1 Satisfying assignment-based approach

As mentioned in §2, a *satisfying assignment* is a sample set of values that satisfy a logical formula. In the context of document recovery, it is a sample set of input bytes which take the program along the execution path characterised by the given PC. The simplest approach for creating document recovery candidates is to take the PC for the given alternative path and ask the constraint solver for a satisfying assignment to all the potentially corrupt bytes. Then, the values returned by the solver need to be assigned to the corresponding bytes in the file, while the rest of the bytes remain unchanged.

The problem with this simple approach is that we do not have any influence on the values returned by the solver. The values in the satisfying assignment will obey the PC, but they will not necessarily be similar to the original file. However, this issue is alleviated by our strategy of modifying only the bytes identified by taint tracking. In our experiments, the largest number of potentially corrupt bytes identified in the taint tracking phase was 25. Even if all these bytes are changed in the satisfying assignment returned by the constraint solver, they represent only a small fraction of the total file size.

3.4.2 Precise algorithm

For more precision when creating recovery candidates, we also used an algorithm that tries to minimise the number of changed bytes. The algorithm, already implemented by KLEE and ZESTI in another context, iterates over each potentially corrupt byte b_i in the original file, and asks the solver whether its value v_i obeys the new PC. If that is the case, the constraint $b_i = v_i$ is added to the PC. This process can be expensive if many bytes are identified as potentially corrupt, because it involves one solver query for each such byte. On the other hand, if only a relatively small number of bytes are identified as potentially corrupt, as is the case in our experiments, the overhead of the algorithm is acceptable.

3.4.3 Constraint independence optimisation

To optimise the candidate creation algorithm, we remark that after selecting one of the alternative execution paths, it is only the last constraint that makes some of the byte values invalid (*i.e.*, not feasible on the alternative execution path), because the prefixes of the PCs for the original and alternative paths are the same for the two executions. One optimisation that we use in both candidate creation algorithms is to eliminate all the potentially corrupt bytes that do not interact with the last constraint, either directly or indirectly. In other words, we compute the transitive closure of the constraints dependent on the last constraint; any bytes not involved in the transitive closure can be left unchanged in the file. For example, in our toy program in Figure 1, when we follow the alternative path with PC $(x \geq \text{'5'}) \wedge \neg(y \geq \text{'5'})$, the transitive closure consists solely of the last constraint. Since the first byte in the file (stored in variable *x*) is not involved, we know we can leave its value unchanged. Note that this optimisation makes use of the constraint independence pass introduced in [5].

3.5 Candidate validation

After a recovery candidate is created, DOCOVERY still needs to check that execution does not hit another bug. DOCOVERY guarantees that the candidate follows a different path but does not guarantee that it is correct.

We re-execute the program natively as soon as the candidate document becomes available, and check that it behaves correctly. As discussed in the overview, we currently only check that the program does not crash, does not terminate with a non-zero return code, and does not output an error message. To account for bugs that cause the program to hang, we set a timeout. If the timeout is exceeded, we discard the candidate document.

3.6 Analysing candidate similarity

The output of DOCOVERY is a set of recovery candidates that do not crash the program and retain a high level of similarity to the broken document. In order to quantify the similarity of the candidates to the broken document, we employ a metric based on the Levenshtein distance.

The *Levenshtein distance* [17], also known as *edit distance*, is a byte-level similarity metric. The two documents are treated as a sequence of bytes, and their edit distance is the minimum number of byte insertions, deletions or substitutions required to change one document into the other.

As discussed in the overview, the degree of similarity between two documents ultimately depends on the specific file type, but in the absence of a high-level document specification, the edit distance can be a useful proxy for many document types. Users are presented with all the recovery candidates generated within a certain amount of time ordered by their edit distance to the broken document. It is up to them to choose the most appropriate candidate.

4. IMPLEMENTATION

We implemented our prototype system on top of the KLEE [4] symbolic execution engine, using the concolic execution functionality from ZESTI [20]. Each analysed program first needs to be compiled into LLVM [16] bitcode, the representation on which KLEE operates. We currently use a modified version of the `whole-program-llvm`[2] script for the compilation process. LLVM was configured to emit bitcode in which *switch* statements are represented as *if* statements.

[2]`https://github.com/travitch/whole-program-llvm`

Table 3: Benchmarks used to evaluate DOCOVERY.

Application	Document type	Document size	
		App specific	Kilobytes
`pr`	plain text	4.4K–1080K chars	4.4–1080
`pine`	MBOX mailbox	5–320 e-mails	13–2314
`dwarfdump`	executable	test, ln, tac, dwarf-dump, dwarfdump2	62–1088
`readelf`	object file	strstrnocase.o print_types.o naming.o print_abbrevs.o dwconf.o print_frames.o	54–1615

Since KLEE keeps track of all memory objects in the program, it can detect memory problems at the point where they occur. The ability to detect errors early makes it possible to avoid running the program after the error occurs and simplifies the recovery process.

DOCOVERY collects alternative execution paths in a FIFO queue of a configurable size. As the execution proceeds, paths associated with more shallow branch points are removed from the queue and deeper paths are inserted.

5. EVALUATION

Our experimental evaluation aims to provide information regarding the scalability of the technique, the types of programs and documents to which it is applicable, and its limitations.

We used an HP Compaq 8200 Elite CMT machine with an Intel Core i7-2600 CPU at 3.4GHz, 16GB of RAM and a Seagate ST500DM002-1BD14 SATA HDD, running Ubuntu 12.04 LTS.

5.1 Benchmarks

We evaluate our prototype system on four C applications processing various types of documents (see Table 3). We have manually injected faults into documents to trigger previously reported bugs in each of the applications. So the bugs are real but the broken documents are synthetic. For each application, we tested documents of various sizes. Below, we provide a brief overview of each application and the corresponding bugs triggered by broken documents.

`pr`: a paginating tool. This tool is part of the `GNU Coreutils` application suite, available in virtually all Linux distributions. It has 1,712 lines of code (LOC) and is linked against the `libcoreutils` library, which has about 39,600 LOC.[3] `pr` takes as an input a text file and paginates it according to a user specification. To create broken documents, we have used a buffer overflow bug in `pr` reported in [4], present in `Coreutils 6.10`. We trigger the error by inserting a 'buggy' sequence of one hundred backspace characters followed by a tabulation. The command line that we used is `pr -e300 file.txt`.

We have tested recovery on synthesised text documents of sizes varying from 4.4KB to 1.1MB. The documents were a sequence of single paragraphs of *Lorem ipsum*[4] text, 80 characters wide. We injected the 'buggy' sequence at the end of each file.

`pine`: an e-mail client. `pine` is a text-mode e-mail client, which is orders of magnitude bigger than `pr`, at around 220,000 LOC; we statically linked `pine` with the `ncurses 5.9` library, which has about 76,000 LOC. We used a known bug[5] that causes older versions of `pine` to crash while displaying a message index containing a specially crafted e-mail; we used version 4.44 in our experiments. The bug is triggered by a *From* field in which the sender e-mail address has multiple escaped double quote characters placed before the @ character. The exact address that we used is `"\" ... "@host.fubar`, with the highlighted part repeated 30 times. This bug may prevent users from accessing their mailbox, despite the fact that all the other e-mail messages are valid.

In order to successfully recover `pine` mailboxes with our prototype, we instrumented the program to indicate when it is done loading a mailbox. Annotating the program to detect that the document was loaded correctly would be needed in other similar interactive applications. We used the following command line arguments to jump directly to a message index displaying the problematic e-mail and used a custom `.pinerc` configuration file to set additional options: `pine -i -p ./.pinerc -n[6|11|21|41|81|161|321]` (the numbers correspond to the corrupt e-mail number).

Recovery was tested with mailboxes of sizes varying from 5 to 320 e-mails. We injected the 'buggy' message at the end of each mailbox. The injected message did not have a subject and a body. The size of the corresponding mailbox file ranged from 13KB to 2.3MB.

`dwarfdump`: a debug information display tool. This utility is used to read and display debugging information stored in DWARF format. In contrast to the first two applications, `dwarfdump` operates on binary files. We used version `20110612` of `dwarfdump`[6] and version `0.8.13` of its `libelf`[7] dependency.

`dwarfdump` has about 12,500 LOC and the linked libraries `libdwarf` and `libelf` have about 21,700 and 5,500 LOC respectively. The bug that we used for our benchmark was reported in [20] and is triggered by setting a specific byte in the file to zero, which causes a division-by-zero error during a sanity check done by the program. We used two different command lines: `dwarfdump file` and `dwarfdump -r file`.

We tested recovery with five executable files, listed in Table 3, with sizes ranging from approximately 60KB to 1MB, in which we injected the fault described before.

`readelf`: an ELF file display tool. The purpose of the `readelf` utility is to dump information about object files in the ELF format. We used `readelf` from `binutils` revision `6e09faca`, whose size is 11,510 LOC; the size of the linked `libiberty` library is 28,900 LOC. The bug that we used for our tests was reported in [21]. It is triggered when the user wants to print out the contents of the `.debug_ranges` section of the 'buggy' file and the relocation offset of the section is negative. This results in a buffer overflow and a program crash. We used the command line `readelf -wR file`.

We have prepared six object files extracted from the `dwarfdump2` utility of the `libdwarf-code` package (the same revision used for the `dwarfdump` tests), with sizes ranging from approximately 54KB to 1.5MB. We injected faults into these files by setting the relocation offset to `0xFD FF FF FF FF FF FF FF`.

5.2 Taint tracking results

Table 4 summarises the results of running the taint tracking algorithm described in §3.1. The third column shows the total number of potentially corrupt bytes identified by the algorithm, while the last column lists those bytes, using angle brackets to denote a comment.

`pr`. The taint tracking algorithm selected a single byte, namely the tabulation character (hexadecimal value `0x09`) from the 'buggy' sequence.

[3]All measurements are done using the `cloc` tool.
[4]`https://code.google.com/p/lorem-ipsum-generator`
[5]`http://www.securityfocus.com/bid/6120`
[6]`git://git.code.sf.net/p/libdwarf/code`
[7]`http://www.mr511.de/software`

Table 4: Taint tracking results.

Application	Document	Number of potentially corrupt bytes	Potentially corrupt bytes
`pr`	4.4K–1080K chars	1	`...0x08<repeated 100 times>0x09EOF`
`pine`	5–320 e-mails	25	`"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"...@host.fubar`
`dwarfdump`	test, ln, tac, dwarf-dump, dwarfdump2	2	`GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3<11 bytes>..<hex 0x00 00>`

`readelf`	strstrnocase.o
	print_types.o
	naming.o
	print_abbrevs.o
	dwconf.o
	print_frames.o
		16	`0x007B38:40 01 00 00 00 00 00 00...0x00D3B8:FD FF FF FF FF FF FF FF`
			`0x00C178:50 09 00 00 00 00 00 00...0x0180B8:FD FF FF FF FF FF FF FF`
			`0x0159C0:80 18 00 00 00 00 00 00...0x02EC90:FD FF FF FF FF FF FF FF`
			`0x02AF58:C0 3E 00 00 00 00 00 00...0x068848:FD FF FF FF FF FF FF FF`
			`0x052278:F0 B3 00 00 00 00 00 00...0x0C2230:FD FF FF FF FF FF FF FF`
			`0x099BC8:A0 06 01 00 00 00 00 00...0x16E758:FD FF FF FF FF FF FF FF`

`pine`. In total, 25 characters were selected, namely the first 25 escaped quotation marks in the 'buggy' e-mail address (the address is 73 characters long).

`dwarfdump`. Two consecutive bytes were identified, the first of which was the byte that we previously corrupted. For readability purposes, we provide in Table 4 an ASCII representation of the relevant file fragment, although the files are binary.

`readelf`. The taint tracking algorithm selected two ranges of eight bytes each, with the second range containing the 'buggy' value. In Table 4, we show the bytes using the format
`offset:potentially corrupt bytes in hexadecimal`.

It is interesting to note that for all benchmarks, the same number of bytes were selected by the taint tracking algorithm regardless of the file size, *e.g.*, for `pine` it selected 25 bytes regardless of the mailbox size, which varied from 5 to 320 e-mails.

5.3 Performance

We have performed scalability tests for our benchmarks, checking the recovery time for files of various sizes, starting with a small document, and roughly doubling its size in each experiment. In this section, we first comment on the results obtained using the simple satisfying assignment-based candidate creation algorithm; the performance of the precise algorithm is evaluated separately at the end of the section.

DOCOVERY was configured to collect up to 500 alternative paths, and to use a 5s timeout for the candidate validation step. The timeout value was established by measuring the time needed to load the documents being considered, which took typically less than 3.5s.

Figures 6–10 present the results. The first bar of each measurement provides the *interpretation time* as a reference point. It corresponds to running the application under DOCOVERY without marking the file as symbolic, until the bug is hit. This value represents the time spent by the symbolic execution engine interpreting LLVM bitcode. The second and the third bars represent taint tracking and alternative paths collection times. The fourth bar is the cumulative value of bars two and three and represents the time needed to prepare for recovery. The fifth bar represents the time needed to generate the first recovery candidate, measured from the beginning of the recovery process. Finally, the last bar shows the total recovery time, *i.e.*, the time needed to exhaust all collected alternative paths.

Overall, the results show that the time spent on taint tracking and alternative paths collection dominates the time needed to create the first recovery candidate. Once these stages are completed, generating recovery candidates is relatively quick.

`pr`. The performance results for `pr` are presented in Figure 6. The time taken to generate the first recovery candidate was only a few minutes for document sizes of up to 271KB, and more than half an hour for the largest document tested. The time needed to create the first recovery candidate once the alternative paths were collected was less than one second for all document sizes.

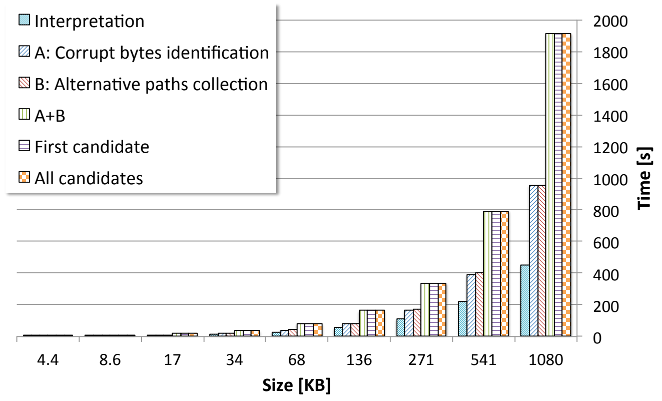

Figure 6: `pr` performance measurements.

`pine`. The results for `pine` are presented in Figure 7. As for `pr`, the time taken to generate the first recovery candidate varied from only a few minutes to around half an hour. The time needed to create a first suitable recovery candidate once all the necessary tainting and alternative paths information was collected was between 4.2s and 114.3s. The higher than expected total recovery time for the 20 e-mail mailbox was the result of encountering multiple timeouts during candidate validation.

`dwarfdump`. For `dwarfdump`, we have tested two scenarios: the first one using the `-r` option, which makes the execution reach the bug faster, and the second without this option. The results are presented in Figures 8 and 9. The length of the execution until the bug is reached can have a significant impact on performance: the overall time taken to create the first recovery candidate was under 40s in the first scenario, and between a few minutes to almost an hour in the second scenario. In both cases, DOCOVERY needed under one second to create the first recovery candidate, once the alternative paths were collected.

`readelf`. The results for `readelf` are presented in Figure 10. As with the other benchmarks, once the alternative paths were collected, the overall time needed to create the first candidate was small, at under one second for all files.

Memory consumption. We measured memory consumption of DOCOVERY itself, excluding the memory needed for the constraint solver and native execution. The memory footprint of DOCOVERY was not a problem: the maximum memory consumption among all benchmarks never exceeded 880MB.

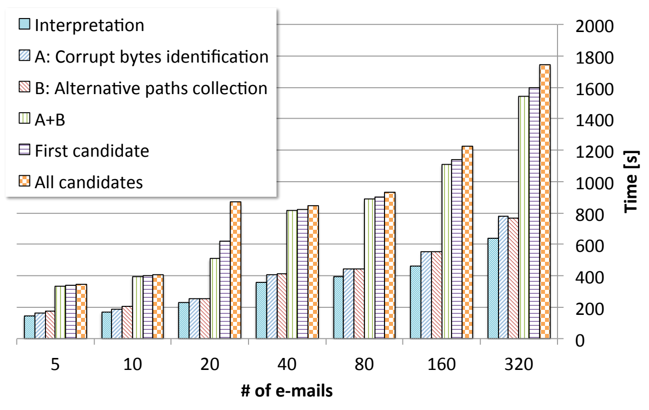

Figure 7: `pine` performance measurements.

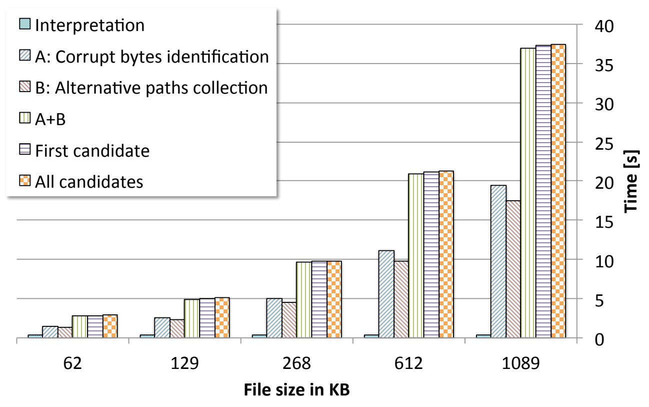

Figure 8: `dwarfdump` performance measurements (with -r).

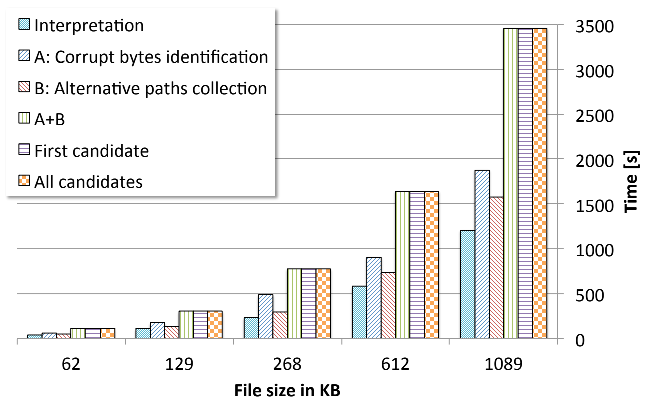

Figure 9: `dwarfdump` performance measurements (without -r).

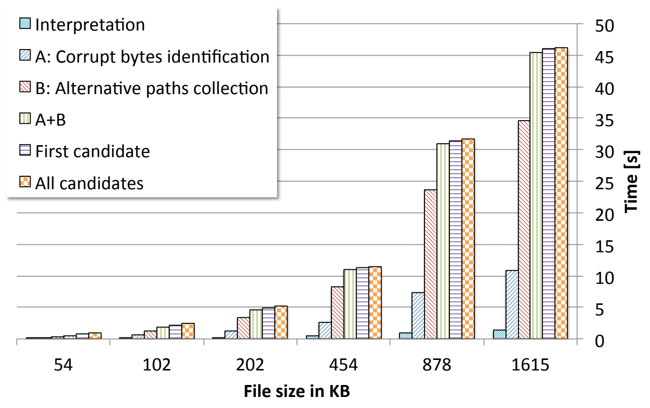

Figure 10: `readelf` performance measurements.

Performance of the precise candidate creation algorithm. Running times for the simple and the precise candidate creation algorithms were similar—usually the difference between the two algorithms was below 10% to generate the first recovery candidate, and it took up to approximately 53% longer for the precise algorithm in the worst case. Similar differences hold for the time to generate all recovery candidates. Many times, the precise algorithm performed better than the simple one. The performance differences are small because taint tracking identifies only a small number of bytes, and thus the precise algorithm only needs to perform a limited number of additional calls to the constraint solver. For `dwarfdump`, the number of calls for the precise algorithm was the same as for the simple one.

5.4 Recovery quality

In this section we evaluate the quality of the created recovery candidates, based on the edit distance to the original document, and our manual analysis of the recovered documents. Tables 5 and 6 present the number of candidates created per document for each benchmark, as well as the minimum and the maximum edit distance between a candidate and the broken document, across all document sizes. Remember that by design DOCOVERY only changes the bytes identified as potentially corrupt in the taint tracking step, so the maximum number is at most equal to the number of potentially corrupt bytes.

`pr`. DOCOVERY created three recovery candidates for each of the input documents regardless of the document size and the candidate creation algorithm used. All candidates differed in one byte from the broken file, namely the last byte of the 'buggy' sequence. Table 7 shows some examples of changes to the last byte's value.[8]

We performed a manual examination of the results by running `pr` natively on the created files. All of the candidate documents seem to be printed out correctly including some text that we later added after the changed byte. We did not notice any significant difference between the candidates. The return code of the application was always 0. Interestingly, the `Null` character `0x00` present in the first candidate does not cause truncation of the output.

`pine`. When the simple candidate creation algorithm was used, DOCOVERY created between 25 and 27 candidates which had one or more of the escaped quotation marks changed. Some examples of recovery candidates are shown in Table 8. When multiple characters were changed, the value for the first one was either the control character `0x0E` (*Candidate A*) or backslash (*Candidate B*), and the rest of the new values were control characters `0x0E`; the edit distance for these candidates ranged between 15 and 24. When only one character was changed, the new value was the `Null` character `0x00` (*Candidate C*).

When the precise candidate creation algorithm was used, 8 candidates were created for each of the tested mailboxes. All the candidates differed in one byte from the original file, which was always changed to the `Null` character in the recovered file (*Candidate C*). The candidates created by using the precise algorithm were the

[8] Note that in general, the exact values chosen for the modified bytes can differ across runs, due to non-determinism in the constraint solver and KLEE.

Table 5: Simple algorithm: an overview of recovery results.

Benchmark	Candidates per document	Edit distance	
		Min	Max
`pr`	3	1	1
`pine`	25-27	1	24
`dwarfdump`	2	1	1
`readelf`	1-3	2	8

Table 6: Precise algorithm: an overview of recovery results.

Benchmark	Candidates per document	Edit distance	
		Min	Max
`pr`	3	1	1
`pine`	8	1	1
`dwarfdump`	2	1	1
`readelf`	1-3	1	8

same as the candidates with edit distance 1 created by the simple algorithm.

We verified all candidates manually by running `pine` natively with each of them and opening the recovered message after the mailbox was loaded; this discarded between 4 and 6 candidates for each mailbox size, which caused a crash when the message was opened.

The recovery candidates for `pine` illustrate the fact that the edit distance is not always the best similarity metric. The mailboxes with edit distance 1 were the ones which had one of the address bytes changed to a `Null` character. As a result, the address in the 'buggy' e-mail was truncated in such candidates. For candidates with a larger edit distance, the new values were such that it was still possible to read the domain of the e-mail address.

`dwarfdump`. For each of the files, two recovery candidates were created, both differing in one byte from the original file. The same files were produced by using the simple and the precise candidate creation algorithms. Table 9 shows the byte changes performed.

We verified the files by running `dwarfdump` natively on them. For all file sizes, the execution of the first candidate finished with a zero return code producing the debug information dump, while the execution of the second candidate ended with an error (return code 1).

We also compared the output of the program for the first candidate with that produced with the initial valid file and there were differences.

Finally, since the recovered files are executables, we manually confirmed that all the created candidates can still be executed by running them and checking their basic functionality.

`readelf`. Three candidates were created for *strstrnocase.o* and a single candidate for the other files. Table 10 presents the bytes changed for the *strstrnocase.o* file, using the simple (*Candidates A, B, C* with edit distances 8, 8 and 2) and the precise candidate creation algorithms (*Candidates D, E, F* with edit distances 1, 8, 2).

We checked the created files by running `readelf` natively on them. The application return code was always 0. We compared the program output with the output produced when running the corresponding correct files. All the candidates, except for candidates B, C, E and F of *strstrnocase.o* produced a warning and had the beginning offset in the first printed row different than their corresponding correct counterparts. The amount of output printed out was otherwise the same as for the correct files. Candidates B, C, E and F for *strstrnocase.o* produced almost no output: for candidate B

Table 7: Examples of recovery candidates for `pr`.

Document	'Buggy' sequence (hex)
Original	`...09`
Candidate A	`...00`
Candidate B	`...0C`
Candidate C	`...0A`

Table 8: Examples of recovery candidates for `pine`.

Document	'Buggy' sequence
Original	`..."{\"}<30x>"@host...`
Candidate A	`..."{\"}<10x>{\0x0E}<15x>{\"}<5x>"@host...`
Candidate B	`..."\"\\{\0x0E}<23x>{\"}<5x>"@host...`
Candidate C	`..."{\"}<24x>\0x00{\"}<5x>"@host...`

(and identical candidate E) an error was printed and for the candidate C (and identical candidate F) the application did not find debug data in the object file.

Since in `readelf` the tested documents are object files, we tried to link each of the created candidates against the original program (`dwarfdump2`). None of the files could be linked successfully. This illustrates the fact that DOCOVERY is application specific: that is, in this case the recovery is done with respect to `readelf` and not with respect to the linking and executing the code in the object files.

For `pr` and `pine`, we injected the 'buggy' sequences at the end of the file because we expect the two applications to process their input files sequentially. We also verified how DOCOVERY performs for these applications when the 'buggy' bytes are injected before the end of the file. For `pr`, we used the 541KB file and injected the sequence after approximately 269KB. For `pine`, we used the mailbox with 160 emails and injected the 'buggy' message after the 80th email. The taint tracking algorithm selected the same number of bytes as in the previous experiments. For `pr`, the same number of recovery candidates were produced, while for `pine`, DOCOVERY produced 23 candidates, out of which two were rejected because they caused `pine` to crash. The running times were about 10% higher for `pr`, and about 40% higher for `pine`, compared to the corresponding documents of about half the size (271KB and 80 e-mails, respectively) that had the fault injected in the end.

6. LIMITATIONS

In §5 we demonstrated that our prototype can recover documents for medium-sized applications. We illustrated recovery on unstructured text data (`pr`), semi-structured *MBOX* file format (`pine`) and binary files (`dwarfdump`, `readelf`).

To show an example in which our approach does not work, let us consider a bug present in version 1.0 of the `MuPDF` PDF processing library.[9] The bug is triggered in the parser's own recovery procedure, when it tries to discover the locations of object definitions in the file. This procedure starts after encountering an invalid offset to a cross-reference table that normally stores offsets of objects. If one of the object numbers is corrupted, it can overflow to a negative value and pass a sanity check inside the parser, which subsequently leads to a buffer overflow and a crash of the program. We tried to recover from this bug, but we were unable to create a recovery candidate with the presented approach.

The challenge associated with this corrupted PDF file, and more generally, with complex structured file formats are dependencies between bytes in the document. In our example, even if we are

[9]`http://www.exploit-db.com/exploits/23246`

Table 9: Examples of recovery candidates for `dwarfdump`.

Document	‘Buggy’ sequence (hex)
Original	...0000...
Candidate A	...0100...
Candidate B	...0001...

Table 10: Examples of recovery candidates for `readelf`, *strstrnocase.o* file, simple (A–C) and precise (D–F) candidate creation algorithm.

Document	‘Buggy’ sequence (hex)
Original	...40 01 00 00 00 00 00 00...FD FF FF FF FF FF FF FF...
Candidate A	...40 01 00 00 00 00 00 00...F0 01 00 00 00 00 00 80...
Candidate B	...FE FF FF FF FF FF FF FF...FD FF FF FF FF FF FF FF...
Candidate C	...00 00 00 00 00 00 00 00...FD FF FF FF FF FF FF FF...
Candidate D	...40 01 00 00 00 00 00 00...FD FF FF FF FF FF FF 00...
Candidate E	...FE FF FF FF FF FF FF FF...FD FF FF FF FF FF FF FF...
Candidate F	...00 00 00 00 00 00 00 00...FD FF FF FF FF FF FF FF...

able to identify the incorrect object number as the source of the problem, how to choose a new object number on the alternative path remains an open question. In order to successfully recover this PDF document in the mentioned scenario, we would need to change the corrupt bytes to a valid object number, because this object can be referenced by other objects by its number. By changing the number to an arbitrary value, we may violate these inter-object dependencies and thus create an invalid document. At the place where the program crashes, we do not have enough constraints to choose the right value. One possible solution would be to implement an iterative strategy, in which we first recover from the original bug and if the recovered document hits another bug, we use that document as the new input to the recovery process.

Furthermore, our current approach and prototype have the following limitations:

- Our approach cannot add or remove bytes during recovery. It is limited to mutating existing bytes. This does not mean it is limited to recovering from corruptions that mutate bytes. It can sometimes recover documents that are corrupted by adding bytes, as shown in our `pr` and `pine` benchmarks.
- Our approach cannot yet recover documents with several independent mutations.
- We only handle bugs that result in generic errors such as program crashes, buffer overflows, and error return values.
- The current prototype does not support file modifications during loading.
- The current prototype requires C source code, although our approach could in principle work directly on binaries.

7. RELATED WORK

The work most closely related to DOCOVERY is that of Rinard et al. [19, 24], who introduced the idea of *input rectification*. At a high-level, their SOAP system starts with a learning stage, in which the system learns a set of constraints characterising typical inputs—for example, it may learn that the height of an image is typically less than a certain value. Then, in the rectification stage, inputs not obeying the constraints are modified to do so—*e.g.*, the image might be truncated to have its height within the limit inferred during the learning phase. SOAP was successfully used to rectify various media files like WAV or JPEG files that were crashing corresponding applications. Compared to DOCOVERY, SOAP scales to much larger applications and document sizes, because enforcing those constraints is much cheaper than using symbolic execution to explore alternative paths. SOAP also does not require direct access to the application code or binary. The key differences are that SOAP requires a specification of the input format and a training set to learn acceptable values for input fields. DOCOVERY does not require a training set or any knowledge about the input format.

Demsky and Rinard [12] propose an approach for repairing data structures starting from a manually-written formal specification. This approach has been successful in recovering errors, *e.g.*, in a corrupted *ext2* file system and in a Microsoft Word file, but the main downside is that it requires developers to write a formal specification for their data structures.

Similarly to DOCOVERY, prior research on vulnerability signature generation, such as Vigilante [9], Bouncer [8] and ShieldGen [10], proposes to monitor the execution of malicious inputs to infer byte-level constraints that characterise such inputs. However, unlike DOCOVERY, the goal is not to modify but to discard such inputs.

An alternative approach to changing the document in order to avoid an error is to change the code of the application itself. Automatic generation of code patches can be accomplished using symbolic execution [22], invariant enforcement [23], genetic programming [2, 30] and various heuristics [18, 27, 28].

Taint analysis and symbolic execution are well-known program analysis techniques that have been used in a variety of contexts, such as testing [1, 13, 14, 21, 29], debugging [7] and attack generation [3, 31], just to name a few. DOCOVERY uses and adapts these techniques in the context of document recovery.

8. CONCLUSION

One of the most visible and frustrating defects of consumer software are application crashes and errors experienced while trying to load a document. In this paper, we presented DOCOVERY, a novel document recovery technique based on symbolic execution. Unlike prior approaches, DOCOVERY does not require any knowledge about the document format, which makes it applicable to a wide range of applications. The key idea behind DOCOVERY is to start from the code path executed by the broken document, explore alternative paths that avoid the error, and finally make small changes to the document in order to force the application to follow one of these alternative paths.

We applied DOCOVERY to popular applications such as the e-mail client `pine`, the pagination tool `pr` and the binary file utilities `dwarfdump` and `readelf`, for which it has managed to recover documents of various sizes, typically within minutes. While these preliminary results are encouraging, we still need to overcome several important challenges in order to make DOCOVERY practical: dealing with complex structured formats, supporting the addition and removal of bytes, and scaling to larger documents.

9. ACKNOWLEDGEMENTS

We would like to thank Paul Dan Marinescu, Hilary Glasman-Deal and the anonymous reviewers for their valuable help and comments. This research is generously supported by Microsoft Research through a PhD Scholarship and by EPSRC through an Early-Career Fellowship.

10. REFERENCES

[1] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *Proc. of*

the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07), Mar.-Apr. 2007.

[2] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proc. of the IEEE World Congress on Computational Intelligence (WCCI'08)*, June 2008.

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Proc. of the 18th Network and Distributed System Security Symposium (NDSS'11)*, Feb. 2011.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Oct.-Nov. 2006.

[6] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the Association for Computing Machinery (CACM)*, 56(2):82–90, 2013.

[7] J. Clause and A. Orso. PENUMBRA: Automatically indentifying failure-relevant inputs using dynamic tainting. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)*, July 2009.

[8] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct. 2007.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of Internet worms. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Oct. 2005.

[10] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'07)*, May 2007.

[11] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the Association for Computing Machinery (CACM)*, 54(9):69–77, Sept. 2011.

[12] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Proc. of the 18th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, Oct. 2003.

[13] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, May 2009.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[15] J. C. King. Symbolic execution and program testing. *Communications of the Association for Computing Machinery (CACM)*, 19(7):385–394, 1976.

[16] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*, Mar. 2004.

[17] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[18] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proc. of the 2nd ASIAN ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*, Mar. 2007.

[19] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, June 2012.

[20] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, June 2012.

[21] P. D. Marinescu and C. Cadar. KATCH: High-coverage testing of software patches. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[22] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[23] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, Oct. 2009.

[24] M. Rinard. Living in the comfort zone. In *Proc. of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, Oct. 2007.

[25] E. J. Schwartz, T. Avgerinos, and B. David. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'10)*, May 2010.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sept. 2005.

[27] A. Smirnov and T. Chiueh. Automatic patch generation for buffer overflow attacks. In *Proc. of the 3rd International Symposium on Information Assurance and Security (IAS'07)*, Aug. 2007.

[28] M. Süßkraut and C. Fetzer. Automatically finding and patching bad error handling. In *Proc. of the 6th European Dependable Computing Conference (EDCC'06)*, Oct. 2006.

[29] N. Tillmann and J. De Halleux. Pex: white box test generation for .NET. In *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, Apr. 2008.

[30] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, May 2009.

[31] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'06)*, May 2006.