Response to Reviewers' Comments

First of all, we want to thank all the reviewers for their valuable comments and suggestions, which are very helpful for improving the paper. We answered the reviewers' questions as follows (review comments in RED and authors' reply in blue) and will incorporate all of their suggestions in this revised version. We have open-sourced the EUNOMIA and the AES's interpreter on https://github.com/EUNOMIA-DEV/EUNOMIA (follow the double-blind criteria). We are actively maintaining the main repository and will keep improving the tool in the future.

==========

Reviewer A

==========

Reviewer A Comment 1

"I would have appreciated a bit more detail about why the authors think unoptimized EUNOMIA outperforms the other tools. They seem to imply that a poor memory model and lack of external functional emulation is responsible for EUNOMIA's success over EOSafe, but I wonder about KLEE and Angr which are widely-used, state-of-the-art tools. I found it very surprising that even without AES files, EUNOMIA outperforms these two engines."

Authors' Reply:

Besides AES and the interval-based algorithm, we also have some engineering efforts that optimize the speed of symbolic execution. If the constraints are satisfied, KLEE and Angr fork as many states as possible. This will lead to an exponential increment of the number of paths (states). Moreover, because KLEE and Angr adopt a stack to store states generated during the symbolic execution, its exponential increment will finally hit the stack space limit, leading to unexpected termination.

In EUNOMIA, we optimized these two factors by: shifting part of the burden related to path explosion to the SMT-solver, like the *ite* statement used in the fully symbolic memory (see Section 6); and adopting priority queue during the execution to reach the user most concerned state as soon as possible.

Reviewer A Comment 2

"The timeout of 1-minute does seem quite short for the evaluation of triggering the logic bombs. From the original work proposing the benchmark, it seems that the results of Angr improve with a 300 second timeout, but still remain behind EUNOMIA."

Authors' Reply:

Thanks for raising the concern! We have tried to use 300s as the time limit for Angr. Eunomia still outperforms it. Specifically, only one timeout logic bomb (under 60s) is successfully triggered (under 300s), which traverses a list structure in C++. The rest of the results remain unchanged. We did not

include them in the paper because they didn't differ too much from the 60s time limit. We will add the numbers under the 300s time limit in our next version.

Reviewer A Comment 3

"The authors do present AES as a key contribution of the paper though, so I would have expected to see a few more experiments highlighting it. How does EUNOMIA perform in detecting the six reported vulnerabilities without AES? Is AES really key to this result?"

Authors' Reply:

Thanks for raising the concern! We emphasize that AES plays a vital role in EUNOMIA for detecting vulnerabilities due to its effectiveness and flexibility. The AES files can not only be regarded as a vulnerability detector (similar to the vulnerability detector plugins of other symbolic execution engines) but also alleviate the path explosion problem during the analysis process. Please refer to the response to **Reviewer B Comment 2** for more details.

Reviewer A Comment 4

"While I found the introduction of the paper to be well-presented, I did find parts of the technical section hard to follow. I think the presentation of advice and pilot and the following instrumentation description could be made clearer. Perhaps having a running example through section 4.1 might be helpful."

Authors' Reply:

Thanks for the advice! We will add more self-contained background knowledge in the beginning of these sections. We will also extend the motivating example to Section 4.1 to illustrate how the *pilot* and *advice* work, and Section 4.2 to illustrate the following instrumentation process.

==========

Reviewer B

==========

Reviewer B Comment 1

"How does your DSL compare to manual insertion of assertions and assumptions, in particular the halt and cons statements?"

Authors' Reply:

Compared to manual insertion of assertions and assumptions, adopting AES files has several advantages. First, we want to emphasize that EUNOMIA supports both modes, while adopting AES files can keep the integrity of to-be-analyzed software. Second, in terms of *halt* statements, i.e., terminating the symbolic execution process on demand, currently widely-adopted tools require hook on a statement (like *assert("exit")*). In this way, if the program happens to have an assertion in other places, the analysis process would unexpectedly exit, which may lead to false negatives. Third, in terms of *constraint* statements, ***the AES files can bring in more flexibility, as the user's concerned constraints can be bound on multiple statements simultaneously***. For example, if developers want to place integer overflow checks on add operators, they need to insert the corresponding constraints after

each statement with add operators. However, adopting AES files, they can bind the constraints on every add operators using a single statement as: *call($add) {sum = $LHS + $RHS; halt = (sum < $LHS || sum < $RHS);}*.

Reviewer B Comment 2

"Do you have any experimental data on benchmarks other than the "logic bombs"? I do not find the experimental evaluation convincing. The benchmarks appear to be contrived (based on the fact that the loop benchmarks rely on the Collatz function), although the paper gives little detail (e.g., size data for the benchmarks). Also, no details are given about how the other tools are called."

Authors' Reply:

Thanks for raising the concern. We omitted some details in RQ2 due to the page limit, which may have caused some misunderstanding.

Note that the logic bomb is only one of the benchmarks we have used. In RQ2, we also evaluate EUNOMIA with several widely-used open-source C and Go projects. To be specific, they either own thousands of stars in GitHub, or are incorporated in official releases like Ubuntu and Go. These open-source projects have 700~16,000 lines of code. KLEE and Angr both fail within six hours because of the path explosion problem on 3 out of 6 samples written in C.

We use EUNOMIA (with/without AES) to detect six realistic bugs. Experiment results show that EUNOMIA with AES can successfully detect the six vulnerabilities in our evaluation with a significant reduction in consumed time. We use the divide-by-zero vulnerability in Go Images as an example, whose entry point is illustrated in the following code:

```
// width and height of the image
var w, h int
fmt.Scanf("%d", &w)
fmt.Scanf("%d", &h)

img := image.NewRGBA(image.Rect(0, 0, w, h))
var buf bytes.Buffer
// encode the img into the buf
if err := tiff.Encode(&buf, img, nil); err != nil {
    fmt.Printf("Unexpected encode error.")
}
// decode the buf, divide-by-zero in its implementation
if _, err := tiff.Decode(&buf); err != nil {
    fmt.Printf("decode error w=%d h=%d", w, h)
}
```

The program first initiates the width and the height of an image as *w* and *h*. Then, the image will be packed into an *image.NewRGBA* structure. The *Encode* function will encode the *img* into a pointer, *buf*, which points to a bytes array. Last, the *buf* will be passed to the *Decode*, where the divide-by-zero on the image width may occur. As we can see, the *buf* is a symbolic pointer that points to a variable length of buffer whose length depends on the image's width and height. If these two variables are given by users, the number of possible combination of these two variables are 2^256*2^256. Thus,

without AES files guidance, EUNOMIA cannot trigger the divide-by-zero vulnerability within 6 hours.

However, generating test input in corner cases follows the intuition of program testing. In this example, we can use AES files to limit the value of *w* and *h* as 0 and INT_MAX - 1 as follows:

```
div-zero {
    cuse(w) or cuse(h) {constraint = (w == 0 or w == INT_MAX - 1)
and (h == 0 or h == INT_MAX - 1);}
    call($div) {halt = ($RHS == 0);}
}
```

In this way, the search space is significantly reduced to 2*2 out of 2^256*2^256. Therefore, with the assistance of the AES file, EUNOMIA can detect the divide-by-zero vulnerability within 20 minutes.

We use the logic bomb benchmark in our evaluation because it is a well-designed benchmark for symbolic execution evaluation [57]. It covers challenging problems for symbolic execution, which can better show the effectiveness of a symbolic execution from the perspective of technique design. Combining the designed micro-benchmark in RQ1 and the realistic projects in RQ2 evaluates EUNOMIA from different angles.

For all our evaluation, we configure KLEE and Angr as follows:

```
Compile C source code files for Angr:
> clang++ -Iinclude -Lbuild -o <out-file> -xc++ - -lutils -lpthread -lcrypto -lm

Run Angr:
# Identical to the command in [57]

Compilie C source code files for KLEE:
> clang -Iinclude -Lbuild -Wno-unused-parameter -emit-llvm -o <bc-file> -c -g <c-file> -lpthread
-lutils -lcrypto -lm

Run KLEE:
> klee --libc=uclibc --posix-runtime --search=dfs -solver-backend=z3 exit-on-error-type <bc-file>
```

Note that we configure the traversal algorithm of KLEE as DFS instead of random to make the performance results consistent across multiple times of experiments. Also, we changed the backend SMT-solver as z3, which is identical to Angr and EUNOMIA, to eliminate the possible bias.

Reviewer B Comment 3

"The concept of intervals is not explained clearly. What exactly is meant by a "subgraph" (i.e., is this just a subset of nodes?)? In the example shown in Figure 3, why is {2,3,4} an interval? It does not contain any closed edges (since the nodes 5 and 6 are not in the subgraph), so shouldn't the interval then also include 1?"

Authors' Reply:
Thanks for raising this issue. We will revise this part, adding some background knowledge to make it self-contained. The concept of "interval" in the control flow graph (CFG) is first raised in [1], while the concept of subgraph follows traditional graph theory. In a CFG, a subgraph is defined as a graph whose nodes and edges are subsets of those of the original CFG.

In Figure 3, if we follow the definition of the closed path and the generation of intervals (both of which are introduced in the first paragraph of Section 5.1), node 1 should be inserted into *H* firstly (step 1). One of node 2's immediate predecessors, node 6, is not in any of the intervals already. Thus node 2 should be inserted into *H* instead of *I(1)* (step 3). Once node 2 is popped from *H* (step 2), which is ahead of the interval *I(2)*, all the immediate predecessors of both nodes 3 and 4 are already in an interval (node 2), thus both of them should be appended into the *I(2)* (step 2). However, node 7, as one of the immediate predecessors of node 5, is not in the interval already. Thus node 5 cannot be appended into the *I(2)*. That is why *I(2)* is only composed of nodes 2, 3, and 4. The process of generation of *I(5)* is similar.

We will improve the writing quality of these definitions and the generation process of intervals.

Reviewer B Comment 4
"I cannot follow the description of Algorithm 1. In particular, I cannot understand where exactly the intervals come into play. This needs to be illustrated better by means of an example."

Authors' Reply:
Thanks for the suggestion! Algorithm 1 actually contains several key concepts in this paper, i.e., interval traversal, state pruning and state reordering.

In terms of the interval traversal, the variable *heads* in Line 1 represent all heads of intervals, such as nodes 1, 2, and 5 in Figure 3. Moreover, in Line 22, we examine whether the current node and its predecessor are located in different intervals. If they are (i.e., inter-interval), the DFS-like traversal will be performed at Line 23 to 28 by storing and restoring visited states. Otherwise, i.e., intra-interval, Lines 23 to 28 will be jumped over, and the loop at Line 20 will iterate all possible states on the given state and basic block, similarly to the BFS-like traversal.

We will try our best to better illustrate the algorithm, for example, we will rename the key variables (like *heads*), and add necessary comments. We will also add a running example of how the interval traversal is implemented based on Figure 3 to make the whole process clearer.

Reviewer B Comment 5
"The paper would also benefit from a careful proof-editing pass.
- *For example, the text in 5.2 is written entirely in conjunctive, which makes it sound strange.*
- *Many of the cross-references seem to refer to the wrong leve (i.e., section rather than subsection).*
- *Some wording is not appropriate (e.g., line 462, "which he hates").*"

Authors' Reply:
Thanks for pointing out these writing issues! We will first improve these pointed out issues as follows:

- Remove unnecessary conjunctive adverbs and reorganize the logic between some statements;
- Recheck all the labels and cross-reference to make the reference in accordance with the statements;
- Improve all these inappropriate words.

Besides, we will conduct several passes of proof-reading to improve our writing quality.

Reviewer B Comment 6

"The paper claims that the algorithm is correct because it collapses into a BFS search if there is no Aes file, but due to the presentation issues, it remains unclear to me why the BFS search is correct."

Authors' Reply:

Thanks for the comment! As we replied to **Reviewer B Comment 4**, the variable *heads* at Line 1 and the loop at Line 20 jointly play a role in interval traversal on the CFG. If the AES file is not provided, Lines 24, 27, and 29 can be omitted because the variable *vars* is not given. More importantly, without the definition of *prior* in the AES file, the priority queue at Line 30 will collapse to a normal FIFO queue. After filtering possible states according to the satisfiability at Line 19, the loop at Line 20 will iterate all possible states from a queue. Consequently, the whole process collapses into a BFS search. We will improve the corresponding writing issue and make this statement clearer by giving a concrete example based on Figure 3.

Reviewer B Comment 7

"Most of the improvements of Eunomia over Klee and Angr comes from two categories, floating point numbers and external function calls. Given that at least Klee is very widely used, I would have expected some discussion of these failures. I would also expect to see some comparison on real-world systems."

Authors' Reply:

We agree that part of the improvements on EUNOMIA's performance on the logic bomb benchamrk can be attributed to the floating point numbers and external function calls categories. Though KLEE is widely used, its original version adopts the strategy that steps in each of the library functions during the symbolic execution. As for memory modeling, KLEE would fork states satisfied on the constraints of the symbolic pointer as many as possible, which result in exponential increment on the path number. Thus for those external functions that extensively manipulate memory area and calculate on floating numbers, EUNOMIA outperforms KLEE due to its memory modeling strategy and external library functions' emulation. The correctness and effectiveness of both of these two strategies are proved by EOSafe [26].

As for the real-world systems, we provided evaluation of real systems in RQ2. It includes six open-source projects in C and Go. These projects are widely-adopted and have 700~16,000 lines of code. We repeated six vulnerabilities in RQ2. Please check the response to the **Reviewer B Comment 2** for more details.

==========

Reviewer C

==========

Reviewer C Comment 1

"The framework would have some research value, once released as open-source tool"

Authors' Reply:

The release of EUNOMIA, as well as the AES's interpreter, has been on our agenda for a long time. We have open sourced it on: https://github.com/EUNOMIA-DEV/EUNOMIA. We are planning to actively maintain it, and update the latest experimental results (including the reproduction and exploiting of vulnerabilities in benchmarks and real-world softwares).

Reviewer C Comment 2

"I am a bit confused about the motivating example in Section 2. The authors said that it is often hard for users to specify the concrete program locations and thus the guidance 1 is challenging to utilize. However, in Listing 2 (the same example), concrete numbers of expected inner and outer loops are also provided by the users, who are supposed to be familiar with the code, which is also hard to apply in practice."

Authors' Reply:

Thanks for the question! We assume that programmers are familiar with the code.

First, in practice, the number of interesting locations can be many. Specifying all of them one by one could be tedious and error-prone. Assume a project has multiple return statements like *return -1*, directly or indirectly. It is easy for programmers to overlook one of the returns if they need to be specified one by one. Second, the loop bound specification is used to accelerate path exploration. EUNOMIA can still correctly check user-specified properties without this information. A practical scenario is that programmers have knowledge about a subset of loops. Therefore, they just specify loop bounds for the loop they know. Note that this is different from specifying all interesting points. For example, in the case of return statement specification, missing one statement can lead to missing vulnerabilities.

Reviewer C Comment 3

"The motivation of introducing def-use of variables (rather than the line numbers) for specifying a program location is not clear (Section 4.1)."

Authors' Reply:

Thanks for raising the concern! Compared to directly specifying line numbers, the key point of locating variables through def-use is that it can match multiple lines of code. For example, with the line number specifying, if the user hopes to perform a check once before each invocation of function *foo*, she has to look through the source code from beginning to end and specify each line where the *foo* is called. However, with def-use AES file, she can just write: *call(foo)*, following which the constraints can be appended.

Reviewer C Comment 4

"Section 4.2 is hard to follow. It is strongly suggested to give some examples to help explain those materials in this section."

Authors' Reply:

Thanks for the suggestion. As we replied to the **Reviewer A Comment 4**, we will extend the motivating example to Section 4 to make the instrumentation process more clear.

Reviewer C Comment 5

"It is not clear about how much benefit the interval (of CFG) design brings in practice. I am not familiar with the state of the practice for handling the code granularity in terms of symbolic execution. Some discussion about the related work is needed."

Authors' Reply:

Thanks for the suggestion! As the Algorithm 1 illustrates, the introduction of interval makes the flexible state pruning and reordering that are designated by the AES files possible. The above process has to be supported by importing auxiliary variables on interval-level (e.g., the *rounds_i* in listing 2), instead of fine granularity as basic blocks. We will discuss the handling of the code granularity in terms of symbolic execution in Section 9.

Reviewer C Comment 6

"The experiments for RQ2 is not very convincing to demonstrate the effectiveness of Eunomia in the real-world setting, as only limited numbers of applications and vulnerabilities are considered (the selection criterion is not clear)."

Authors' Reply:

Thanks for raising the concern. We argue that these 6 samples illustrated in RQ2 are representative, and are able to reflect the effectiveness of EUNOMIA with AES files. Please refer to the response to the **Reviewer B Comment 2** for more details.

Reviewer C Comment 7

"The formal syntax and semantics of the advice keyword in Aes is not clear."

Authors' Reply:

We will improve our description about the "advice" keyword.

Reviewer C Comment 8

"Fig. 1 is not utilized well (seems neither explained nor referenced by any algorithms to help illustration)."

Authors' Reply:

Actually Fig.1 is referenced in the "Our Solution" of Section 2. We admit, however, the figure lacks enough explanation. We will add the legend and a set of descriptions on this figure, which will help the reader in understanding the motivating example.