Towards Automatically Generating a Sound and Complete Dataset for Evaluating Static Analysis Tools

Aravind Machiry, Nilo Redini, Eric Gustafson, Hojjat Aghakhani,
Christopher Kruegel, and Giovanni Vigna
{machiry, nredini, edg, hojjat, chris, vigna}@cs.ucsb.edu
University of California, Santa Barbara

***Abstract*—Binary static analysis has seen a recent surge in interest, due to a rise in analysis targets for which no other method is appropriate, such as, embedded firmware. This has led to the proposal of a number of binary static analysis tools and techniques, handling various kinds of programs, and answering different research questions. While static analyis tools that focus on binaries inherit the undecidability of static analysis, they bring with them other challenges, particularly in dealing with the aliasing of code and data pointers. These tools may tackle these challenges in different ways, but unfortunately, there is currently no concrete means of comparing their effectiveness at solving these central, problem-independent aspects of static analysis.**

In this paper, we propose a new method for creating a dataset of real-world programs, paired with the ground truth for static analysis. Our approach involves the injection of synthetic "facts" into a set of open-source programs, consisting of new variables and their possible values. The analyses' goal is then to evaluate the possible values of these facts at certain program points. As the facts are injected randomly within an arbitrarily-large set of programs, the kinds of data flows that can be measured are widely-varied in size and complexity. We implemented this idea as a prototype system, AUTOFACTS, and used it to create a ground truth dataset of 29 programs, with various types and number of facts, resulting in a total of 2,088 binaries (with 72 versions for each program). To our knowledge, this is the first dataset aimed at the problem-independent evaluation of static analysis tools, and we contribute all code and the dataset itself to the community as open-source.

I. Introduction

The static analysis of programs has been studied at length throughout computing's history. While many of these advances focus on the analysis of source code or other high-level abstractions, recent work has focused on the analysis of compiled binary programs [42], [47], [51]. This could be mainly attributed to the explosion in the number of Internet of Things (IoT) devices or in general embedded devices, where, most often, only the binary firmware is available [2].

Static analysis is often preferred [46] for analyzing these firmware binaries. Although, dynamic analysis tools exist, they either need real devices [56] or considerable engineering effort [12], [14], [15], [17].

Binary static analysis tools need to handle several challenges introduced by the underlying architecture. A sound static analysis technique for binaries might suffer from low precision, as it has to solve various challenges to handle pointers [7], [20], type inference [10], etc. Current tools and techniques provide a best-effort static analysis [19], [23], [31], [41], [47] using reasonable heuristics [16], [25], [30], [55], which compromise soundness for precision. The effectiveness of these heuristics is highly dependent on the dataset and its architecture [3], [8]. Furthermore, ground truth on the analysis results is not always available for binaries. This makes the fair comparison of different binary static analysis techniques a challenging problem.

However, there is currently no general benchmark or measurement methodology to assess and compare the core analysis performance of these approaches. Central to this is the need for ground truth; we need a sound and complete set of known *facts*, such as possible values for a variable at a given program point, or the set of aliases a pointer can have at a given time. However, we cannot merely generate these from existing source code, while also being sound and complete, as this is, again, undecidable.

In this paper, we propose a new methodology for measuring the performance of binary static analysis tools. This approach is based around the automated injection of synthetic *facts*–a set of possible variable definitions–into a target program. When analyzing these test programs, the analysis frameworks will be forced to evaluate the possible content of these facts at various points in the program, and the results can be compared with ground-truth. By injecting our facts into existing programs, we side-step the need for complete ground-truth from the original program, and also create a very realistic dataset. The logic of the program itself is intact and unaltered, and the analysis systems have to deal with various degrees of complexity inherent to the program.

We implemented these ideas into a prototype, called AUTOFACTS, on top of the LLVM framework [33], which can be used to generate datasets for any source code, and for any target architecture supported by LLVM. We demonstrate AUTOFACTS on 29 programs by injecting various types of facts resulting in a total of 2,088 binaries, and discuss its effectiveness at making a diverse, challenging dataset for the evaluation

Workshop on Binary Analysis Research (BAR) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-58-4
https://dx.doi.org/10.14722/bar.2019.23090
www.ndss-symposium.org

of static analysis tools.

In summary, our contributions are as follows:

- We propose a new method of evaluating static analysis tools, by creating ground-truth using known "facts" injected into programs.
- We present AUTOFACTS, a prototype implementing these ideas. AUTOFACTS can transform the source code of programs to add facts in an automated fashion.
- We create a dataset of 2,088 binaries with 29 programs using the code of popular open-source programs, along with the ground truth data for various data and functions pointers, and contribute this to the community [1].

II. Background and Motivation

Static analysis, as applied to compiled binary programs, takes on a different set of challenges from those of source code or high-level bytecode. This is due, in part, to the loss of information in the compiled binary. For instance, information regarding program variables [6] and their types [35], as well as memory aliasing [5], [24] are removed during the compilation process.

Static Data Flow Analysis or Data Flow Analysis (DFA) [38] is one of the common forms of static analysis used for various security applications. Most of the DFA-based techniques depend on the availability of accurate points-to information and a complete Control-Flow Graph (CFG) [1] of the program.

Points-to analysis is a known hard problem, and it has been the subject of research for multiple decades. Although recovering the CFG is relatively easy for source-code-based analysis, binary analysis techniques still struggle to produce an accurate CFG in the presence of indirect jumps or function calls [20], [31], [51]. Most of the existing techniques are based on heuristics [3], [16]. Although some techniques perform relatively well compared to others, the lack of proper sound and complete ground truth hinders an accurate evaluation of these approaches.

A relatively easy way to generate good ground truth is to use source-code-based analysis. One could use existing source-code-based techniques to produce a sound result [4], [57]. However, binary analysis techniques tend to compromise soundness for precision [47]. Furthermore, as the source code-based techniques cannot be sound and complete, it is possible that a binary analysis technique could produce results on a binary that are more precise than a source-based-analysis on the corresponding sources. This makes the comparison against source code-based ground truth a trickier problem. For instance, if a binary analysis technique misses a target for an indirect call, it is hard to say whether it is correct (because the binary analysis is more precise) or wrong (it legitimately missed a possible target).

What we need is a technique to generate a sound and complete dataset, so that we can accurately evaluate the effectiveness of static binary analysis techniques.

[1] github.com/ucsb-seclab/autofacts

III. Related Work

To the best of our knowledge, AUTOFACTS is the first dataset released for evaluating the static analysis techniques being used in binary analysis tools, independent of the ultimate goal for which these tools are being used. In this section, we briefly survey works have been done in the related area, though with a different focus than our work.

Evaluating bug detection tools. Wilander et al. implemented a testbed of 44 function calls in C to investigate the effectiveness of five publicly available static bug detection tools [52]. One year later, they created 20 different buffer overflow attacks to evaluate four publicly available tools for dynamic buffer overflow detection [53]. Zitser et al. [59] manually assembled a ground truth corpus of source code examples containing 14 exploitable buffer overflow vulnerabilities found in three open-source software to evaluate five static buffer overflow detectors. Zhivich et al. [58] later used the same 14 vulnerabilities to compare a few dynamic buffer overflow detection tools. As a follow-up analysis to the Zitser's study [59], Kratkiewicz et al. [32] generated more diagnostic test cases, a corpus of 291 small C programs, to determine specific strengths and weaknesses of the tools.

Later, a much bigger and more useful public database for the evaluation of bug detection tools was generated by the NIST project Software Assurance Metrics And Tool Evaluation (SAMATE) [29]. The major part of their evaluation corpus is represented by Juliet [21], a collection of 86,864 synthetic C and Java programs including 118 different CVEs. More recently, Shiraishi et al. [45] created 638 variations of the 51 different types of defects that can lead to runtime exceptions and should be detected by static analysis tools, and they conducted a quantitative analysis of commercial static analysis tools by using their test suites. Later, Shoshitaishvili et al. [47] reproduced many existing approaches in binary vulnerability analysis using their proposed binary analysis framework, angr, to compare the effectiveness of binary vulnerability analysis techniques by evaluating them against a dataset created by DARPA [40].

Dolan-Gavitt et al. [18] leveraged a taint-analysis-based technique for producing ground-truth corpora of vulnerable programs by automatically injecting bugs into the program source code. Using their technique, they released the LAVA dataset, which can be used for the evaluation of both static and dynamic analysis tools. Recently, Bonett et al. [9] proposed a mutation-based framework that systematically evaluates Android static analysis tools. They first create security operators that reflect the goals of the tools being analyzed, e.g., buffer overflow detection. These security operators then are inserted into Android apps, which results in the creation of multiple mutant versions of the original app. Ultimately, the static analysis tools can be evaluated against the injected mutants. While this is somewhat similar to AUTOFACTS, this work operates on high-level bytecode, and focuses specifically on security-related findings.

General evaluation of program analysis techniques. While related works mainly evaluate the existing program analysis tools by focusing on the final goal for which these tools are being used, the research community has started to invest effort into ensuring the preciseness and reliability of program

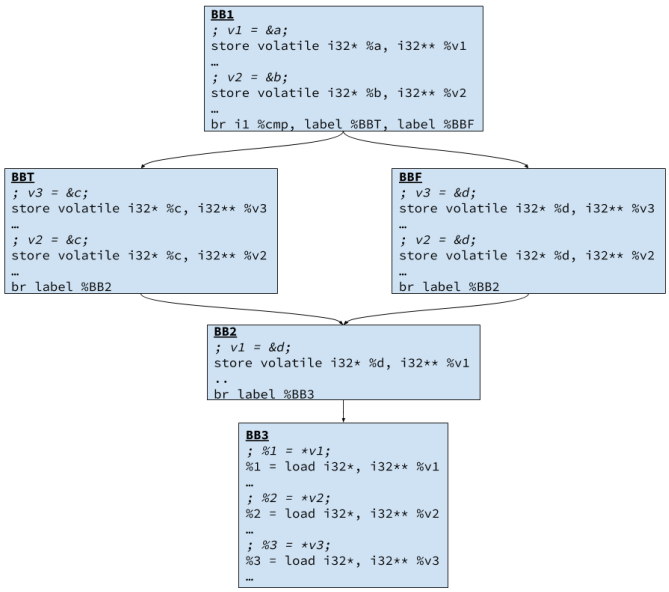

Fig. 1. Control flow graph (CFG) of a sample function illustrating various fact generation techniques of AUTOFACTS

analysis tools, regardless of the problem they are trying to solve [11], [28].

As an example, Kapus et al. [28] adopted compiler testing techniques to automatically find errors in symbolic execution engines. They managed to find 20 major bugs in three widely-used symbolic execution engines. Moreover, Wu et al. [54] presented a system for evaluating different pointer alias analysis implementations by validating the results of pointer alias analysis tools against the pointer values that were observed at runtime execution. They found that many pointer alias analysis implementations wrongly state that two pointers never alias, whereas they actually do during the execution.

While these techniques do operate in a general setting, like AUTOFACTS, none of them operate on binary programs while guaranteeing a sound and complete result.

IV. Methodology

In this section, we present the design of AUTOFACTS, our system for inserting facts in binary programs such that we know the sound and complete result expected from a static data flow analysis (DFA).

A. Sensitivities

There are certain design choices any DFA makes to have a tradeoff between precision and performance. These design choices are often called *sensitivities*, and a few of the commonly used sensitivities are summarized in Table IV. We use the notation, S_f, S_p, S_{c_n} to indicate that a DFA is *flow*, *path* or *n-context* sensitive, respectively. We use the call-stack as the context of a function. An n-context sensitive analysis uses the call-stack as the context, but if the call-stack has more than n elements, it only considers the *top* n elements in the call-stack. Furthermore, we append the subscripts to indicate multiple-sensitivities i.e., we use S_{fp} to indicate that a DFA is both flow and path sensitive. We use S_ϕ to indicate the analysis has no sensitivity. For instance, Steensgaard pointer analysis [49] is a S_ϕ analysis.

AUTOFACTS generates facts such that any DFA with certain sensitivities should produce the corresponding results. Specifically, a S_x fact consists of a set of definitions (d) and a use (u) of a variable with a result set (r), such that any DFA with S_x sensitivity should have r as the value set for the variable at the point of use, i.e., u.

B. Corpus Generation

Without *a priori* specification of the required program logic, generating complete and realistic programs automatically is a hard problem [37]. Moreover, the generated programs could be small and may not be representative of real-world programs. We follow the trend of the recent vulnerability injection works [18], [43], and resort to inserting facts into existing programs.

C. Facts generation

Generating a fact involves inserting a set of definitions of a variable, which we call the target variable (v), and a use of the variable at different points in the control-flow graph (CFG) of the whole program. The program points where the definitions and use of a fact should be inserted depends on the sensitivity of the fact. The result set of a fact includes a set of values or a *set of* set of values.

1) S_ϕ fact: An S_ϕ fact does not consider the control flow of the program, as such the definitions and use can be inserted at any arbitrary locations in the CFG.

- **Definitions** (d): The definitions can be at any arbitrary set of basic blocks, $d = \{BB_1, BB_2, ..., BB_n\}$, in the CFG of the program to the same variable (v).
- **Use** (u): The use of the variable (v) can be at any arbitrary basic block BB_u in the CFG.
- **Result set** (r): The result set is a set of values used at *all* the locations where the variable is defined.

Any analysis that has no sensitivity or that it is S_ϕ sensitive should have r as the value set for v at u. This is because, if the analysis is S_ϕ sensitivity, it should not respect the control flow, i.e., it has no kill sets [38] and should consider the effect of all the statements irrespective of the control-flow.

Consider the S_ϕ fact with target variable $v3$ in Figure 1, where the definitions (i.e., `store`) are inserted in $BB1$, BBT, and BBF. The use (i.e., `load` or `%3=*v3`) is in $BB3$, and the expected result set of an S_ϕ analysis at the point of uses $\{\&d, \&c\}$.

2) S_f fact: To generate an S_f fact, we should insert definitions, which can at least test the ability of the analysis to respect the program flow within a function.

- **Definitions** (d): The definitions of the target variable v should be in a list of basic blocks: $d =< BB_1, BB_2, ..., BB_{n-1}, BB_n >$ of a function, such that there exists a linear flow from between them. Formally, any BB_x (x $>$ 1) is a *post dominator* [36] of BB_y (y $\geq$ 1) where y $<$ x.

- **Use** (u): The use of the variable should be in a basic block BB_u such that BB_u is a post-dominator of BB_n.
- **Result set** (r): The result set contains only one value that is used in the definition of basic block BB_n. This is because every definition in a basic block BB_i will overwrite the value defined in any of the previous basic blocks BB_j with $j < i$. Consequently, the only value v can possibly have at BB_n will be the value defined in BB_n.

Any S_f analysis should respect the program flow, and, consequently, should have only one value (i.e., r) in the value set for the variable at the use basic block, i.e., BB_u.

An example of an S_f fact with target variable $v1$ is shown in Figure 1, where the definitions (i.e., `store`) are inserted in the basic blocks $BB1$ and $BB2$, and the use (i.e., `load` or `%1=*v1`) is in $BB3$. The expected result set of an S_f analysis at the point of use, i.e., the possible values of `%1` is $\{\&d\}$, as the definition in $BB2$ overwrites or kills the one in $BB1$.

3) S_p fact: An S_p fact should be able to test the analysis ability to distinguish between different paths in the CFG of a function.

- **Definitions** (d): There should exist one definition of the target variable v in each basic block of a set: $d = \{BB_1, BB_2, ..., BB_n\}$, such that there exists *no* path between any of the basic blocks.
- **Use** (u): The use of the variable should be in a basic block BB_u such that BB_u is a *post-dominator* of all the basic blocks $BB_i \in d$.
- **Result set**(r): The result set contains $|d|$ number of singleton sets, one for each basic block in $BB_i \in d$, and the value in each set is the value used in the definition in the corresponding basic block.

Any S_p analysis will compute a value set for each path in the program. As there is only one definition in every path, the value set computed for v in any path of the program should be a singleton *and* a member of r. There can be cases where BB_u can post-dominate other basic blocks that are not in d, which could be handled by a simple technique (omitted for the reasons of space) by traversing the post-dominator tree.

An example of an S_p fact with target variable $v3$ is shown in Figure 1, with definitions (i.e., `store`) in BBT and BBF, as no path exists between them. The use (i.e., `load` or `%3=*v3`) is in $BB3$, which is a post-dominator of both BBT and BBF. The expected result set of an S_p analysis at the point of use (i.e., the possible values of `%3`) are $\{\{\&c\}, \{\&d\}\}$, one for each path from BBT and BBF respectively.

4) S_{fp} fact: One of the simple ways to generate an S_{fp} fact is to just add a definition in a pre-dominator basic block of a S_p fact. Specifically, first we insert an S_p fact: let the set of basic block where the definitions have been inserted be d, the use basic block be BB_u, and the result set be r. Next, we find a basic block that is a pre-dominator of all the basic blocks in d. Finally, we insert a definition to v in the pre-dominator basic block.

This creates an S_{fp} fact whose result set is also r (i.e., of the original S_p fact). This is because the newly inserted definition is in the pre-dominator basic block, and, therefore it will be killed by a definition in one of the basic blocks of d, as any paths to BB_u must *flow* through one of them.

Similar to an S_p analysis, an S_{fp} analysis will also compute a value set for each path in the program. However, in each path, the control-flow is respected. As explained in Section IV-C3, the value set computed for v in any path of the program should be a singleton *and* a member of r.

The Figure 1 with target variable $v2$ shows an example of an S_{fp} fact. The definitions of $v2$ at basic blocks BBT and BBF, with use at $BB3$, creates a path-sensitivity fact with result set $\{\{\&c\}, \{\&d\}\}$. By inserting a definition at $BB1$ (i.e., a pre-dominator of both BBT and BBF), the fact becomes a flow- and path-sensitive (S_{fp}) fact. The result set remains the same, as the definition in $BB1$ will be killed by the definitions in BBT and BBF.

5) S_{c_n} fact: Generating a context-sensitive fact requires the inverse call-graph R_{cg}, which is a directed acyclic graph with labeled edges, that contains a node for every function in the program and a directed edge from node u to v with label l for every invocation of function u by function v at the call-site l. We construct the R_{cg} from the call graph [44] by inverting the edges and considering only direct calls. Consequently, in R_{cg} every edge has a unique label. We also avoid edges that can lead to cycles.

The S_{c_n} context-sensitive fact should be able to test the analysis ability to differentiate between the different contexts of a function up to length n. An easy way to pass values across different functions is through arguments. Given a function f, we identify all the nodes (or functions) that are n edges away in R_{cg} (we also consider nodes that are less than n edges away only if they have no successors). We call these nodes the target functions t_f. Next, we modify the signature of f to have an additional argument, which is our target variable v. Similarly, we modify all the call sites and functions along each path from nodes in t_f to f to have a new argument, so that a value is passed via the newly inserted argument along the path from each of the functions in t_f to our target function f.

We insert a use of the newly inserted argument v in the first basic block of function f. The result set r would be the set of singletons sets where each set contains a single element i.e., one for each function in t_f. Any S_{c_n} analysis is expected to compute the result set, as it should be able to distinguish between different contexts up to length n. An example of a S_{c_n} fact is shown in Appendix IX-A.

All the other combinations of sensitivities could be inserted by exploiting dominator trees.

V. Implementation

At a high level, AUTOFACTS works by modifying the provided LLVM bitcode by inserting various types of facts. Relying on LLVM bitcode exclusively allows the system to be architecture-independent. Furthermore, LLVM has builtin support to compute dominator trees and all the other instrumentation support needed for the implementation of AUTOFACTS.

A. Type of facts

We believe that pointer analysis [26] and function pointer resolution [31] are two of the main problems in static binary analysis. As such, in our current implementation, we insert only pointer type variables (i.e., data pointers and function pointers). The values are the addresses of the existing variables visible within the scope, i.e., local variables or global variables.

Our current implementation respects language types i.e., all the values used in the definitions have the same type (t), and the used variable will be have the type of a pointer to t. For instance, if we insert a `long**` type variable then all the values used in the corresponding definitions will be of type `long*`.

B. Target variable

To be safe, our facts insertion should not interfere with the existing program logic at runtime. To ensure this, for each fact, we always create a new variable visible within all the basic blocks selected for definitions and use insertions. For instance, to insert an S_f fact, we either create a global or a local variable within the function to which the basic blocks belong. This also ensures that our definitions do not interfere with the existing program logic.

C. Definitions

As values of definitions, we use those available within the scope of all the basic blocks selected for definitions and use. We assume that using existing values within the scope i.e., reading the value of existing variable or getting its address, does not affect the program logic.

The definitions are LLVM `store` instructions of the form: `store * %valToStore, ** %targetVariable`. We make all the stores `volatile` to avoid that compiler optimization technique would remove definitions. These newly inserted store instructions don't affect the program logic as all the changes are to a newly created variable.

D. Use

In most of the cases, the use of the target variable should not affect the program logic as the target variable is newly inserted. However, there are certain cases where the inserted *use* could affect the program logic. For instance, if we have inserted a function pointer variable, the use of which would be an indirect call, then in essence, we insert a call to an existing function. Depending on the function, this could affect the program logic.

To prevent this, we introduce a conditional use guarded by an opaque constant (i.e., opaque predicates [13]). In other words, we insert a conditional statement that is infeasible, but its infeasibility hard to determine statically.

AUTOFACTS is implemented as an LLVM 6.0.0 pass. It takes the whole program bitcode, and inserts a certain (configurable) number of facts randomly over different parts of the program.

E. Inserting a data pointer fact

To insert a fact, we randomly pick a set of definitions and use basic blocks, that satisfy the corresponding sensitivity requirement. For instance, in the case of a flow-sensitive (S_f) fact, as explained in Section IV-C2, for definitions, we select a list of basic blocks such that every basic block post-dominates the previous basic block in the list. For use, we select a basic block that is a common post-dominator of the first and last basic block in the above list. Note that, such a basic block is also a post-dominator of all the basic blocks of the list.

We then randomly pick a set of pointer values of a certain type visible within the scope of the basic blocks. These could be local variables, i.e., `alloca` instructions, or global variables. We ensure that all these values are of the same type $t*$.

Next, we create a local or global pointer variable (random choice) of type `t**` i.e., a pointer to pointer of type `t` visible within all the selected basic blocks.

At each of the definitions basic blocks, we insert a store instruction (Section V-C) of a value into the newly created target variable.

At the basic block containing the use, we insert a condition based on an opaque constant, and, in the body of the condition, we add a `load` instruction from the target variable. To avoid any optimization that might try to remove the load, we further add a call to the `exit` function with the loaded value as an argument. An example of the LLVM instructions that would be added is:

```
%14 = load i32*, i32** %targetVar
%castToInt = ptrtoint i32* %14 to i8
call void @exit(i8 %castToInt)
```

VI. Evaluation

As our base dataset, we selected all the 29 programs from GNU inetutls, compiled them using clang to generate whole program bitcode files [50]. The complete list of the programs is listed in Table V.

Performance: Table I shows the average time in milliseconds AUTOFACTS took to insert various number of data pointer facts (of different sensitivities) to our dataset. The performance was almost same to insert function pointer facts as well.

As expected, inserting an S_ϕ fact takes the least time. Furthermore, inserting an S_f, S_p or S_{fp} fact takes relatively more time as the selected basic blocks need to satisfy certain requirements (Section IV). An interesting thing to note is, insertion of context-sensitive facts (S_c and S_{fc}) takes less time than the path-sensitive fact (S_p). This is because more time is required to find S_p fact compatible basic blocks which require traversals of the dominator tree and CFG.

Complexity of the facts: On average, for flow- (S_f), path- (S_p) and, flow- and path- (S_{fp}) sensitive facts the average distance between the inserted basic blocks and the use basic block is three. Which could be used as a weak proxy for the complexity of the inserted facts. For the context-sensitive facts (S_c and S_{fc}), the length of the context is selected at random.

Number of facts	S_ϕ	S_f	S_p	S_{fp}	S_c	S_{fc}
50	3.69	26.41	88.06	111.23	63.57	64.89
100	7.42	36.75	184.62	223.61	142.21	145.82
150	11.38	213.23	259.25	263.76	226.75	207.92
200	14.65	361.14	388.12	341.12	270.14	286.78
250	18.00	486.84	451.68	455.72	308.60	332.53
300	21.92	515.21	547.62	565.92	390.89	370.85

TABLE I. AVERAGE TIME (IN MILLISECONDS) TAKEN BY AUTOFACTS TO INSERT VARIOUS NUMBER OF FACTS ACROSS DIFFERENT SENSITIVITIES INTO DIFFERENT PROGRAMS IN OUR DATASET.

Number of facts	S_ϕ	S_f	S_p	S_{fp}	S_c	S_{fc}
50	2.22%	1.64%	1.29%	2.1%	2.67%	2.68%
100	4.19%	3.96%	2.42	4.03%	5.2%	5.1%
150	6.07%	6.03%	3.52%	5.71%	7.78	7.78%
200	8.01%	11.48%	4.59%	7.12%	10.36%	10.27%
250	9.97%	14.24%	5.65%	8.94%	12.92%	12.97%
300	11.83%	16.79%	6.68%	10.95%	15.55%	15.82%

TABLE II. AVERAGE PERCENTAGE OF INCREASE IN THE OUTPUT BITCODE SIZE AFTER INSERTING VARIOUS NUMBER OF FACTS ACROSS DIFFERENT SENSITIVITIES INTO DIFFERENT PROGRAMS IN OUR DATASET.

Output size: Table II shows the average percentage increase in the size of output bitcode after inserting various number of data pointer facts. The size increase is similar for inserting function pointer facts as well.

As expected, the size increase is *inversely proportional* to the restrictions on the corresponding fact. This is because, the restrictions decrease the possible places to insert the definitions, consequently the number of definitions will be less and hence the less size. For instance, an S_p fact has more restrictions on the places to insert than S_f and S_ϕ facts. Hence as it is evident from the columns 4, 3 and, 2, S_p facts have less percentage of increase than the S_f and S_ϕ facts.

Table III shows the average percentage increase in the size of executables compiled for x86_64 from the corresponding instrumented bitcode files. As expected, they also follow a similar trend as the bitcode file sizes (Table II). We believe that the size increase would follow a similar trend for the other architectures too.

Result set: The result set for each type of fact is emitted according to the instrumented LLVM bitcode. We intend to tag these using LLVM debug information, which can be used to map the result set on to the binaries.

VII. DISCUSSION

In this section, we discuss the possible issues or limitations of our current approach of the fact generation and possible future work.

- **Realistic facts**: The facts inserted may not be realistic i.e., these may not be representative of the actual usage pattern in real programs. For instance, function pointers are usually defined in a global array and used by indexing into it [51]. Having an evaluation of the common usage patterns in real programs would provide an insight into generating realistic datasets.
- **Additional sensitivities**: There are many other possible sensitivities a static analysis can have, including object sensitivity [48], field sensitivity [38] and choices of heap models i.e., call-site [39] or heap context-sensitivity [34] which could be considered to generate a better dataset.
- **Hardness of the inserted facts**: As our insertion mechanism is random, there could be certain facts that are easy to compute and others that could be more difficult. A metric for hardness of a fact would be helpful for a fine-grained evaluation of the analysis techniques.
- **Non strict types**: As mentioned in Section V-A, AUTOFACTS respects types. Although this is reasonable for well-written programs, there are other programs (e.g., the Linux kernel) that heavily uses unsafe casts [27]. Having non-strict types could help in generating datasets useful for evaluating analysis tools targeting such non-conforming programs.

Number of facts	S_ϕ	S_f	S_p	S_{fp}	S_c	S_{fc}
50	3.92%	2.41%	1.75%	3.27%	3.19%	3.37%
100	7.6%	4.76%	3.17%	6.16%	5.88%	5.83%
150	10.94%	7.79%	4.66%	8.89%	8.55%	8.65%
200	14.52%	14.33%	6.09%	11.01%	11.09	10.98%
250	17.95%	17.65%	7.17%	13.98%	13.47%	13.36%
300	21.39%	20.97%	8.26%	16.86%	15.93%	15.97%

TABLE III. AVERAGE PERCENTAGE OF INCREASE IN THE COMPILED EXECUTABLE (X86_64) SIZE AFTER INSERTING VARIOUS NUMBER OF FACTS ACROSS DIFFERENT SENSITIVITIES INTO DIFFERENT PROGRAMS IN OUR DATASET.

VIII. CONCLUSION

In this work, we presented a means to address the need for sound and complete datasets to evaluate static binary analysis tools. We demonstrated that automatically inserting program facts is a reasonable way to generate such a dataset. Furthermore, we present the methodology and implementation of AUTOFACTS, our system for automatically injecting program facts into LLVM bitcode to generate sound and complete dataset for evaluating DFA-based static analysis techniques. Our preliminary evaluation shows that the proposed method is scalable and fast. In the future, we would like to handle other sensitivities and perform an evaluation of existing binary static analysis tools with our dataset. We believe that approaches such as AUTOFACTSwill allow the community to improve the completeness and usefulness of static analysis tools.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This material is based on research sponsored by the Office of Naval Research (ONR) under grant numbers N00014-17-1-2897, N00014-17-1-2011 and by the National Science Foundation (NSF) under grant number CNS-1704253. This work is also sponsored by a gift from Google's Anti-Abuse group. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of ONR or NSF.

REFERENCES

[1] F. E. Allen, "Control flow analysis," in *Proceedings of the Symposium on Compiler Optimization*, ser. CGO '70, New York, NY, USA, 1970.

[2] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. SP '19, San Jose, CA, USA, 2019.

[3] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proceedings of the IEEE European Symposium on Security and Privacy*, ser. EuroS&P '17, Paris, France, 2017.

[4] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96, New York, NY, USA, 1996.

[5] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proceedings of the International conference on compiler construction*, ser. CGO '04, Barcelona, Spain, 2004.

[6] ——, "Divine: Discovering variables in executables," in *Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI '07, San Diego, CA, USA, 2007.

[7] C. Ballabriga, J. Forget, and G. Lipari, "Abstract interpretation of binary code with memory accesses using polyhedra," *arXiv preprint arXiv:1711.07257*, 2017.

[8] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *Proceedings of USENIX Security Symposium*, ser. SEC '14, San Diego, CA, USA, 2014.

[9] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Discovering flaws in security-focused static analysis tools for android using systematic mutation," in *Proceedings of USENIX Security Symposium*, ser. SEC '18, Baltimore, MD, USA, 2018.

[10] J. Caballero and Z. Lin, "Type inference on executables," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 65, 2016.

[11] C. Cadar and A. F. Donaldson, "Analysing the program analyser," in *Proceedings of the International Conference on Software Engineering Companion*, ser. ICSE '16, Austin, Texas, USA, 2016.

[12] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *Proceedings of the Network and Distributed Systems Security*, ser. NDSS '16, San Diego, CA, USA, 2016.

[13] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the Symposium on Principles of Programming languages*, ser. POPL '98, San Diego, CA, USA, 1998.

[14] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: system-wide security testing of real-world embedded systems software," in *Proceedings of USENIX Security Symposium*, ser. SEC '18, Baltimore, MD, USA, 2018.

[15] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *Proceedings of Asia Conference on Computer and Communications Security*, ser. ASIACCS '16, Xi'an, China, 2016.

[16] A. Di Federico, M. Payer, and G. Agosta, "rev. ng: a unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the International Conference on Compiler Construction*, ser. CGO '17, Austin, Texas, USA, 2017.

[17] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the Program Protection and Reverse Engineering Workshop*, ser. PPREW '15, 2015.

[18] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. SP '16, San Jose, CA, USA, 2016.

[19] C. Eagle, *The IDA pro book*. No Starch Press, 2011.

[20] X. Fan, Y. Sui, X. Liao, and J. Xue, "Boosting the precision of virtual call integrity protection with partial pointer analysis for c++," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '17, Santa Barbara, California, USA, 2017.

[21] C. for Assured Software, "Juliet test suite v1.2 user guide. technical report." National Security Agency, 2012.

[22] GNU, "GNU inetutils," https://www.gnu.org/software/inetutils/.

[23] GrammaTech, "GrammaTech CodeSonar," https://www.grammatech.com/products/codesonar, 2010.

[24] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.

[25] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada, 2018.

[26] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program analysis for software tools and engineering*, ser. PASTE '01, 2001.

[27] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference." in *Proceedings of USENIX Security Symposium*, ser. SEC '04, San Diego, CA, USA, 2004.

[28] T. Kapus and C. Cadar, "Automatic testing of symbolic execution engines via program generation and differential testing," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE '17, Urbana-Champaign, Illinois, USA, 2017.

[29] M. Kass, "Nist software assurance metrics and tool evaluation (samate) project," in *Proceesings of the Workshop on Evaluation of Software Defect Detection Tools*, ser. BUGS '05, Chicago, IL, USA, 2005.

[30] O. Katz, N. Rinetzky, and E. Yahav, "Statistical reconstruction of class hierarchies in binaries," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA, 2018.

[31] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proceedings of the International Conference on Computer Aided Verification*, ser. CAV '08, Princeton, NJ, USA, 2008.

[32] K. Kratkiewicz and R. Lippmann, "Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools," in *Proceesings of the Workshop on Evaluation of Software Defect Detection Tools*, ser. BUGS '05, Chicago, IL, USA, 2005.

[33] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Conference on Compiler Construction*, ser. CGO '04, Palo Alto, California, 2004.

[34] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '07, 2007.

[35] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed Systems Security*, ser. NDSS '11, San Diego, CA, USA, 2011.

[36] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 121–141, 1979.

[37] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, pp. 151–165, 1971.

[38] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[39] E. M. Nystrom, H.-S. Kim, and W.-m. W. Hwu, "Importance of heap specialization in pointer analysis," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program analysis for software tools and engineering*, ser. PASTE '04, 2004.

[40] T. of Bits, "DARPA Challenge Binaries on Linux, OS X, and Windows," https://github.com/trailofbits/cb-multios, 2017.

[41] ——, "Manticore: Symbolic Execution Framework," https://github.com/trailofbits/manticore, 2017.

[42] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: on the security of bootloaders in mobile devices," in *Proceedings of USENIX Security Symposium*, ser. SEC '17, Vancouver, BC, Canada, 2017.

[43] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: challenging bug-finding tools with deep faults," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. FSE '18, Lake Buena Vista, Florida, USA, 2018.

[44] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[45] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *Proceesings of the IEEE International Symposium on Software Reliability Engineering Workshops*, ser. ISSREW '15, Washington DC, USA, 2015.

[46] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *Proceedings of the Network and Distributed Systems Security*, ser. NDSS '15, San Diego, CA, USA, 2015.

[47] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. SP '16, San Jose, CA, USA, 2016.

[48] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 17–30.

[49] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the Symposium on Principles of Programming languages*, ser. POPL '96, St. Petersburg Beach, Florida, USA, 1996.

[50] Y. Sui, "Whole program bitcode file," https://github.com/SVF-tools/SVF/wiki/Install-LLVM-Gold-Plugin-on-Ubuntu.

[51] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *Proceedings of the Network and Distributed Systems Security*, ser. NDSS '17, San Diego, CA, USA, 2017.

[52] J. Wilander and M. Kamkar, "A comparison of publicly available tools for static intrusion prevention," in *7th Nordic Workshop on Secure IT Systems," Towards Secure and Privacy-Enhanced Systems", 7-8 November 2002, Karlstad University, Sweden*. Karlstad University Studies, 2002, p. 68.

[53] ——, "A comparison of publicly available tools for dynamic buffer overflow prevention." in *Proceedings of the Network and Distributed Systems Security*, ser. NDSS '03, San Diego, CA, USA, 2003.

[54] J. Wu, G. Hu, Y. Tang, and J. Yang, "Effective dynamic detection of alias analysis errors," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. FSE '13, Saint Petersburg, Russia, 2013.

[55] X. Yin, S. Liu, L. Liu, and D. Xiao, "Function recognition in stripped binary of embedded devices," *IEEE Access*, 2018.

[56] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *Proceedings of the Network and Distributed Systems Security*, ser. NDSS '14, San Diego, CA, USA, 2014.

[57] S. Zhang and B. G. Ryder, *Complexity of single level function pointer aliasing analysis*. Rutgers University, Department of Computer Science, Laboratory for Computer, 1994.

[58] M. Zhivich and T. Leek, "Dynamic buffer overflow detection," in *Proceesings of the Workshop on Evaluation of Software Defect Detection Tools*, ser. BUGS '05, Chicago, IL, USA, 2005.

[59] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. FSE '04, Newport Beach, CA, USA, 2004.

IX. Appendix

The precision of analysis results with different context sensitivities is incomparable. Consider the second and third row of Table IV, here the result sets of Flow Sensitive and Context Sensitive analysis are incomparable. We cannot say one is more precise than the other as the results are provided along different aspect of the program control flow.

Listing 1. Example to illustrate various sensitivities

```
1:foo() {
2:  bar(4);
3:  bar(5);
4:}
5:bar(int n) {
6:   v = 1;
7:   if(*) {
8:        v = 2;
9:   } else {
10:       v = 3;
11: }
12:}
```

Listing 2. Example of a 2-context sensitive fact inserted in function `foo`

```
main() {
1: bar();
}

bar() {
   int *b;
   ..
   // inserted value
   int **temp = &b;
2: baf(.., temp);
}

baz() {
   int *c;
       ..
   // inserted value
   int **temp = &c;
3: baf(.., temp);
}

// added argument
// to enable propagation
baf(.., int **v) {
   ..
   // propagating argument
4: foo(.., v);
   ..
}

bak() {
   int *a;
   ..
   // inserted value
   int **temp = &a;
5: foo(.., temp);
}

foo(.., int **v) {
   // inserted use
   int *temp = *v;
}
```

A. Context-sensitive fact example

Consider the example in Listing 2 and corresponding inverse call-graph is shown in Figure 2.

Consider that we want to insert a 2-context sensitive fact in the function `foo`. First, referring to the figure, finding all the nodes that are two or less edges (if there are no successors)

TABLE IV. COMMONLY USED SENSITIVITIES ALONG WITH AN EXAMPLE RESULT FOR THE CODE SNIPPET IN Listing 1

Sensitivity	Result Sets	Example
No Sensitvity	One for the whole program.	Possible values of `v` = $\{1, 2, 3\}$
Flow Sensitive	One for each program point.	Possible values of `v` at line 11 = $\{2, 3\}$
Path Sensitive	One for each path of the program.	Possible values of `v` *in path along line numbers 5, 6, 7, 8, 11, 12* = $\{1, 2\}$
Flow and Path Sensitive	One for each path and each point of the program.	Possible values of `v` *in path along line numbers 5, 6, 7, 8, 11, 12* and *at line number 12* = $\{2\}$
k-Context Sensitive	One for each context (up to length k) of a function.	Possible values of `n` with 1-call-site context $< 2 >$ = $\{4\}$

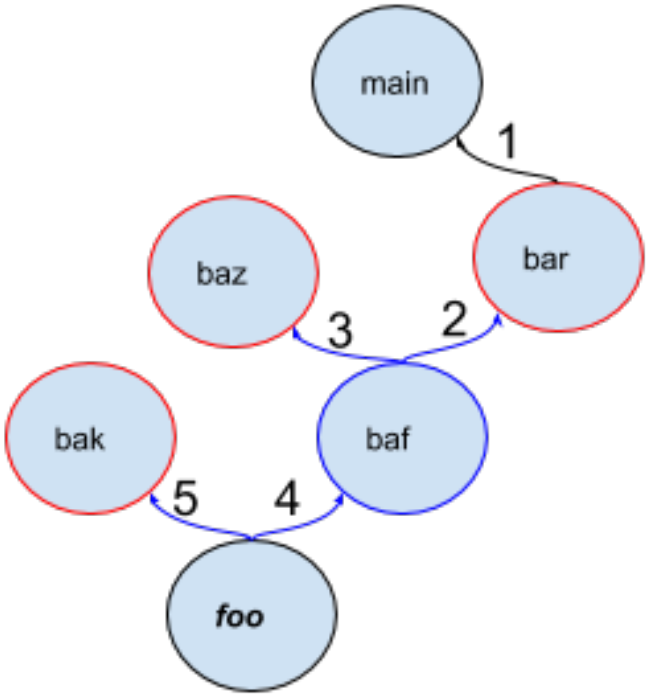

Fig. 2. Inverse Control flow graph (CFG) of the program in Listing 2, where the target functions (t_f) are colored red and the paths to the function `foo` are colored blue.

away gives us: `bar`, `baz` and, `bak`, which are our target functions (t_f).

Second, we change the signature of the function `foo` by adding our target variable i.e., `int **vv` and inserting a corresponding use i.e., `int *temp = *v`.

Next, we modify the call-sites and nodes that are in the path from functions in t_f to `foo`. From (refer the figure), the corresponding call-sites and the functions are $2, 3, 4, 5$ and `baf` respectively. We modify these to propagate a value from functions in t_f to `foo`. As shown in Listing 2, we have modified by adding an additional parameter to `baf` and new argument to the call sites.

Finally, we modify the target functions i.e., t_f to pass a value as an argument to the corresponding call-site.

The result set of this newly inserted fact at the point of use in function `foo` is: $\{b, c, a\}$, one for each of the functions in t_f.

Program Name
dnsdomainname
ftp
ftpd
hostname
identify
ifconfig
inetd
logger
ls
ping
ping6
rcp
rexec
rexecd
rlogin
rlogind
rsh
rshd
syslogd
talk
talkd
telnet
telnetd
tftp
tftpd
thttpd
traceroute
uucpd
whois

TABLE V. NAMES OF THE PROGRAMS IN INETUTILS [22] USED AS OUR DATASET