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 analysis 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 of 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 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
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 function pointers, and contribute this to the community.

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-code-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 that 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
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 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, $S_f$, $S_p$, $S_c_i$, 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 $S_f p$ to indicate that a DFA is both flow and path sensitive. We use $S_\phi$ to indicate the analysis has no sensitivity. For instance, Steensgaard pointer analysis [49] is a $S_\phi$ 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) $S_\phi$ fact: An $S_\phi$ 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 $S_\phi$ sensitive should have v as the value set for v at u. This is because, if the analysis is $S_\phi$ 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 $S_\phi$ fact with target variable v3 in Figure 1, where the definitions (i.e., store) are inserted in $BB_1$, $BB_T$, and $BBF$. The use (i.e., load or %3=*v3) is in $BB_3$, and the expected result set of an $S_\phi$ 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 ≥ 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 $\%1=\star v1$) 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 $\{k,d\}$, 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: $d = \{BB_1, BB_2, \ldots, BB_n\}$, 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 $r$.

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 $\%3=\ast v3$) 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 $\%3$ is $\{k,c\}$, $\{k,d\}$), 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 $\{\{c,d\}, \{k,d\}\}$. 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) $S_c$ 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 $S_c$ 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 $S_{cn}$ 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 $S_{cn}$ 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 built-in 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 \( \text{long} ** \) type variable then all the values used in the corresponding definitions will be of type \( \text{long} * \).

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: \( \text{store} * \%\text{valToStore}, ** \%\text{targetVariable} \). 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., \( \text{alloca} \) 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 \( \text{load} \) instruction from the target variable. To avoid any optimization that might try to remove the load, we further add a call to the \( \text{exit} \) function with the loaded value as an argument. An example of the LLVM instructions that would be added is:

\[
\%14 = \text{load} \ \text{i32}* , \text{i32} ** \%\text{targetVar} \\
\%\text{castToInt} = \text{ptrtoint} \ \text{i32} * \%14 \ \text{to} \ \text{i8} \\
\text{call void @exit(i8 %\text{castToInt})}
\]

VI. Evaluation

As our base dataset, we selected all the 29 programs from GNU inetu1ts, 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 \( S_\phi \) 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.
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 AUTOFACTS will allow the community to improve the completeness and usefulness of static analysis tools.

### 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.

### 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 AUTOFACTS will 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


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

```c
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: }
```

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

```c
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

<table>
<thead>
<tr>
<th>Sensitivity</th>
<th>Result Sets</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Sensitivity</td>
<td>One for the whole program.</td>
<td>Possible values of $v = {1, 2, 3}$</td>
</tr>
<tr>
<td>Flow Sensitive</td>
<td>One for each program point.</td>
<td>Possible values of $v$ at line 11 $= {2, 3}$</td>
</tr>
<tr>
<td>Path Sensitive</td>
<td>One for each path of the program.</td>
<td>Possible values of $v$ in path along line numbers $3, 6, 7, 8, 11, 12$ $= {1, 2}$</td>
</tr>
<tr>
<td>Flow and Path Sensitive</td>
<td>One for each path and each point of the program.</td>
<td>Possible values of $v$ in path along line numbers $3, 6, 7, 8, 11, 12$ and at line number $12$ $= {2}$</td>
</tr>
<tr>
<td>k-Context Sensitive</td>
<td>One for each context (up to length k) of a function.</td>
<td>Possible values of $v$ with 1-call-site context $&lt; 2 &gt; = {4}$</td>
</tr>
</tbody>
</table>

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 $t_f$.

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

<table>
<thead>
<tr>
<th>Program Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>disdomaniname</td>
</tr>
<tr>
<td>ftp</td>
</tr>
<tr>
<td>ftpd</td>
</tr>
<tr>
<td>hostname</td>
</tr>
<tr>
<td>identify</td>
</tr>
<tr>
<td>ifconfig</td>
</tr>
<tr>
<td>inetd</td>
</tr>
<tr>
<td>logger</td>
</tr>
<tr>
<td>ls</td>
</tr>
<tr>
<td>ping</td>
</tr>
<tr>
<td>ping6</td>
</tr>
<tr>
<td>rcp</td>
</tr>
<tr>
<td>reexec</td>
</tr>
<tr>
<td>reexecd</td>
</tr>
<tr>
<td>rlogin</td>
</tr>
<tr>
<td>rlogin</td>
</tr>
<tr>
<td>rtsock</td>
</tr>
<tr>
<td>ssh</td>
</tr>
<tr>
<td>syslogd</td>
</tr>
<tr>
<td>talk</td>
</tr>
<tr>
<td>talkd</td>
</tr>
<tr>
<td>telnet</td>
</tr>
<tr>
<td>telnetd</td>
</tr>
<tr>
<td>tftp</td>
</tr>
<tr>
<td>tftp</td>
</tr>
<tr>
<td>tftp</td>
</tr>
<tr>
<td>traceroute</td>
</tr>
<tr>
<td>uacp</td>
</tr>
<tr>
<td>w年限</td>
</tr>
</tbody>
</table>