Transformation-Aware Symbolic Execution for System Test Generation

Stephen McCamant
Mathias Payer
Dan Caselden
Alex Bazhanyuk
Dawn Song

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-125
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-125.html

June 21, 2013

Transformation-Aware Symbolic Execution for System Test Generation

Stephen McCamant
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN, 55455
Email: mccamant@cs.umn.edu

Mathias Payer, Dan Caselden,
Alex Bazhanyuk, and Dawn Song
Computer Science Division
University of California, Berkeley
Berkeley, CA, 94720
Email: mathias.payer@nebelwelt.net,
dawnsong@cs.berkeley.edu

***Abstract*—A common development task is to take a behavior exercised in a single function (e.g., a failing unit test), and to produce an input to the entire program (a system test) with the same behavior. In security, when the behavior is a potential vulnerability, this is constructing a proof-of-concept exploit. This task is challenging because it requires precise reasoning over an entire program. To automate instances of this task, our approach uses symbolic execution to generate program inputs that undergo transformations before they are used. Using information about the relationship of data structures and transformations in a program, our approach works backward, one transformation at a time, and applies optimized symbolic execution to search for transformation pre-images. Our techniques out-perform standard symbolic execution by several orders of magnitude, and construct exploits against two vulnerable document-processing applications without using source code.**

I. Introduction

Software tests can be broadly classified into unit tests, which test a minimal unit of functionality, and system tests, which test a complete program. A given bug might revealed either by a unit test or a system test, but each type of test has its own advantages and applications. Given a system test that reveals an interesting behavior such as a bug or security vulnerability in a unit, the process of generating a corresponding unit test that reproduces the same bug is mostly mechanical, and previous work discusses techniques for making it completely automatic (e.g., [1], [2]). The opposite problem of producing a system test that illustrates the same behavior as a unit test is more difficult, as illustrated by the observation that in general it may not even be possible: some unit behaviors may be unreachable within a particular larger system. Such system test generation is more challenging to automate because it requires precise reasoning that can span across an entire large program.

One instance of the system test generation problem that is of particular importance occurs when the interesting behavior is a (potential) vulnerability: for instance, a buffer that might potentially be overrun under some circumstances. The system test in this case is a called a proof-of-concept (POC) exploit, since it proves that the potential vulnerability is a real one (but is not weaponized into a full attack). Determining whether an vulnerability is real, and if so producing a proof-of-concept exploit, is a critical task in security analysis, but in current practice it requires significant, expert, manual attention. (In accord with the concerns of this security application area, in this work we concentrate on program behaviors whose incorrectness is easy to identify, such as crashes. We elide the problem, important in other kinds of testing, of constructing an oracle to determine whether a test result is correct. As such our "test generation" task is more explicitly "test input generation".)

Many aspects of a large program can make system test generation challenging, but this paper focuses on a single one: the presence of complex data transformations that lie between the program input and the unit that displays interesting behavior. In order to generate system tests efficiently in the presence of transformations, we describe enhancements that build *transformation-aware symbolic execution* by extending standard symbolic execution. We give search strategies tailored for producing transformation pre-images, and a divide-and-conquer approach that inverts sequences of transformations by inverting each component transformation in reverse sequence. Our approach uses a new program representation called a hybrid information- and control-flow graph (HI-CFG) to represent a program's transformation structure. We implement these new techniques in FuzzBALL, a symbolic execution tool for binary programs that has a number of security applications. In a direct comparison, out techniques improve the performance of symbolic execution over a state-of-the-art but non-transformation-aware tool (KLEE [3]), often by two orders of magnitude or more. In case studies of binary exploit generation, our system generates exploits against two document processing programs without requiring source code.

The primary contributions of this paper are the following:

- We introduce pruning and prioritization search strategies for symbolic execution to allow it to find transformation pre-images more quickly.
- We introduce a divide-and-conquer strategy for pre-image computation of composed transformations based on a HI-CFG.
- We evaluate these improvements by comparing an enhanced version of FuzzBALL with both a vanilla FuzzBALL and another state-of-the-art tool KLEE.

For conceptual completeness, we also summarize the concept of the HI-CFG, and a dynamic analysis approach for building a HI-CFG from instruction-level traces, and describe the security implications of the exploit generation case studies. However these areas are the contributions of another paper which describes them in detail [4]; more information about them is also available in a related technical report [5].

II. Overview

In this section we give a technical overview of our technique for transformation-aware symbolic execution, and in particular define the kinds of transformations for which it is effective.

A. Transformations

Speaking generally, a *transformation* is code within a program whose purpose is to read as input data values in one format, and write as output data values in a different format that in some sense encode roughly the same meaning as the input values. We adopt the perspective that the transformation has one important input and one important output. Generally its behavior will also depend on other aspects of the program state, which we can describe as parameters or meta-data, but the important relationship is between the main input and main output.

Because of the eternal demand for software to be more functional and flexible, one of the main directions of increasing complexity comes in supporting an ever-wider variety of data formats, generally by transforming between them. Examples of such transformations include compression and decompression, encryption and decryption, conversion of text between character sets, and the conversion of documents and images between formats. In fact many modern document formats, such as PDF and Microsoft Office documents, incorporate sub-objects (such as embedded images) that can themselves take many different formats.

Because data transformations are so common, our hypothesis is that we can build more effective tools by taking advantage of the special structure of transformations. In this paper we focus on building transformation-awareness into symbolic execution as applied to the task of finding pre-images for transformations: given an output, finding a corresponding input. This task is motivated by the problem of creating (system) test inputs for programs that use transformations, but we expect that transformation-awareness would improve other applications of symbolic execution as well.

B. Approach Overview

Our approach makes symbolic execution transformation-aware in two ways. First, we propose modifications to the search strategies and related mechanisms of symbolic execution that make it more effective at finding pre-images within the large search space of possible transformation inputs and executions. These techniques allow symbolic execution to prune path prefixes that could not produce the desired pre-image, learn which areas of the search space are productive, and reason efficiently about lookup tables and other constructs implemented with symbolic memory indexing. Second, when a program includes multiple transformations, we observe that it is often much more efficient to take each component transformation separately than to treat them as a single large transformation. To support this approach, we describe analysis techniques to automatically infer the structure of transformations with a program, together with the data structures that hold intermediate values. With this information our system can then compute pre-images for each step of a chain of transformations separately, in reverse sequence.

C. Supported Transformations

We have designed our transformation-awareness mechanisms to target a class of transformations that are common and important across many application domains. However, this choice necessarily excludes some other kinds of transformations for which our techniques would be unhelpful or incompatible. Here we list the key properties of the transformations that our approach supports; later Section VIII discusses directions for future research to further broaden the set of supported transformations.

a) Surjective: A function is *surjective*, right-invertible, or onto its co-domain, if every element in its co-domain is the image under the function of some input. In other words, surjectivity guarantees that a pre-image exists mathematically for any value we pick from the output space. Of course our computational problem of finding a pre-image is only well-posed if a pre-image exists for the particular output we query. But even if we knew through other means that a pre-image exists, having each transformation in a sequence surjective makes the search process easier because it implies that there is possibility of getting stuck or needing to backtrack between transformations. Once we have found a single pre-image of a transformation, we never need to consider that transformation again. A broad class of transformations that decode general data are surjective, including decryption, decompression, and the inverses of transformations that add redundancy or escape certain sequences.

b) Sequential output: We say a transformation has *sequential output* if it produces its output as a sequence of values in a predictable format, so that it is unambiguous at the point a value is produced that it constitutes output, and the value never subsequently changes. (Any transformation could satisfy this property in a trivial way if declared its output to be available only when it finishes executing, but doing so would conflict with the next property.) In other words, a transformation with sequential output never "revises" output values after producing them.

For our techniques to apply, our system must further be able to recognize in an automated way when a value constitutes transformation output. Our current implementation focuses on the case where a transformation produces output by storing values (such as bytes or words) to a contiguous memory area (such as an array). In this case the sequential output property requires that once a location is written, the transformation never later overwrites it with a different value. The common case we observe is also for the written locations to be sequential in these sense of having adjacent and increasing addresses (i.e., writing left-to-right in an array), though this is not a fundamental requirement.

c) Streaming: We say a transformation is *streaming* if its inputs and outputs are interleaved and the transformation keeps only a limited amount of internal state. Streaming is a very implementation approach for transformations, when it applies, because it generally allows a transformation to be implemented efficiently without excess memory requirements. Unix pipelines are a well-known example of a streaming paradigm, and streaming is also very common in multimedia algorithms. The streaming property is important to the effectiveness of our technique's pruning, because a transformation that produces output values early allows our system to discover high in the search tree that a particular input prefix will never produce the desired output.

III. Strategies for preimage search

Our approach uses symbolic execution to search for transformation pre-images. We start by giving a brief introduction to the use of symbolic execution for program exploration (Section III-A), then describe three features and optimizations that make the preimage search efficient: (i) pruning in Section III-B, (ii) prioritization in Section III-C, and (iii) the handling of symbolic memory accesses in Section III-D.

A. Background: Symbolic Exploration

Symbolic execution is a program analysis technique that combines features of dynamic and static analysis by considering families of executions that follow the same execution path. Certain inputs to a program or code fragment under test, rather than taking concrete values such as particular integers, are replaced by symbolic variables. As the code executes, computations on these values produce more complex symbolic expressions, capturing all possible concrete values. When a symbolic expression occurs in a branch condition, we can use a decision procedure such as STP [6] or Z3 [7] to determine which directions for future execution are feasible. Symbolic execution is useful because the symbolic execution of a single path can correspond to a large number of concrete executions, but still be precise: no approximation is involved in computing the symbolic expressions. Another advantage is that arbitrary additional conditions can be conjoined with the formulas (as if they were additional branches in the program) and checked in the same way.

One common application of symbolic execution is to explore within the space of all feasible executions of a code fragment, which we refer to as *symbolic exploration*. Our system explores one execution path at a time, starting with no constraints on the symbolic variables. Each time execution reaches a branch that depends on symbolic values, the tool is free to explore either side of the branch, subject to a feasibility check (ensuring the choice is compatible with the earlier branches taken). The tool keeps track of which sequences of branch choices lead to parts of the execution space that have been fully explored, and avoids them. We describe a further basis it uses for deciding between branch directions below in Section III-C; when all else is equal, it chooses randomly. A more in-depth discussion of symbolic execution techniques is found in the form of a survey [8], or in papers describing tools [3], [9].

If it were allowed to run forever, a symbolic exploration tool would eventually explore every possible execution path through a code fragment. But for all but the smallest fragments, the number of paths is so large (even infinite) that this is not a practical strategy. The key to effective use of symbolic exploration is to guide the search towards execution paths that are more likely to be interesting, which will be the focus of the next two subsections.

B. Search Pruning

The most important technique for reducing the size of the space that must be searched for a preimage is to prune prefixes of the input buffer contents that produce the wrong prefix of the output buffer contents. This is a very common pattern for transformations that can apply to an unbounded input, but keep only a limited-size internal state.

While exploring the execution of the transformation, at each point at which the code writes a value to the output buffer, we check whether it is possible that the written value can be equal to the desired output value at that position. If it cannot be equal, then no extension of the currently explored path could create the desired output, so the search can be pruned at that point, and no extensions are explored. If the values can be equal but are not necessarily equal, such as if the written value is an unconstrained symbolic variable, we add the constraint that they match to the path condition, which can also prune the search space by making some future paths infeasible.

An indication of the power of this pruning is that if the number of reads between consecutive writes is bounded, it will typically reduce the number of paths that can be explored from exponential in the input size to linear in the input size. However applying just this technique the linear factor can still be quite large, which can be further addressed by the two optimizations we describe next.

C. Search Prioritization

Another optimization that can take advantage of checking if the code produces the desired output is to bias the search toward paths that have produced the most correct output values so far. Intuitively, this approach directs the exploration to spend more of its time attempting to extend paths that have already proved promising, as opposed to paths that have not shown results yet. This approach can be described in terms of an utility function for states in the exploration space. For our preimage computation this is a lexicographically ordered pair whose more significant element is the ratio of correct output bytes produced input bytes read, and whose less significant element is the number of correct output values produced by the path up to that point. To implement this approach, our tool records, before each branch point in the search space, maximum utilities of all of the states that have been explored beyond that point. When returning to a branch point that has been visited before, the search will prefer the branch direction with the higher maximum utility.

As in any search process, our search for a preimage has a tension between local and global search. Prioritizing states that have proved effective so far will speed the search if the search space is well behaved. But one would not want to unconditionally prefer the already-proven states, because the search space might have dead ends that appear initially promising, but cannot be extended to give the complete desired output. We do not want a search process that is required to explore such dead ends exhaustively before trying another path. To strike this balance, our system's search prioritization is not absolute. Instead, each time the search reaches a state with a utility-based preference, we flip a biased coin. If the coin comes up heads, we follow the preference, otherwise we fall back to a random choice strategy. For the experiments in this paper, we have set the probability of the biased coin to follow the utility-based preference with probability 0.95. This probability works well for our case studies and follows a greedy strategy that ensures that we first search the depth of the tree before backing up and searching more in the breadth.

D. Symbolic Array Accesses

A final aspect of our use of symbolic execution is not specific to exploring transformations, though it often applies to them. As previously mentioned, an advantage of symbolic execution is that a single symbolic path can correspond to multiple concrete paths. A trade-off with respect to how many concrete paths a symbolic path represents occurs, for instance, when the code uses a load from memory to implement a lookup table.

This trade-off arises when the address value used in a load from memory is symbolic (e.g., `val = array[i];`), so that the load might refer to multiple locations. How should the symbolic execution system implement this load?

One approach, which is the default in our symbolic execution tool, is to treat the selection of which address to load like a multi-way branch. The tool chooses one feasible value for the address (thereby concretizing the value), and continues execution subject to this choice. Later, when it returns to the branch point, it can choose a different feasible value. Since choosing a value for the address effectively makes it concrete, this approach tends to create simple symbolic formulas which can be evaluated efficiently. On the other hand, if many address values are possible, the number of paths to explore can quickly become large.

An alternative approach is to represent all the possible values from the load in the symbolic expression, e.g., Mayhem [10] uses a similar approach. This approach makes sense for the case in which the possible loaded values represent a lookup table, though it need not be limited to that case. The formula representing the symbolic results of the load will itself have the structure of a lookup table (or, equivalently, a circuit representing a ROM). The main limitation is that the number of possible addresses and loaded values cannot be too large, otherwise the symbolic formula become unmanageable.

The trade-off that comes with the lookup table approach is that the number of execution paths to be explored will be much smaller, but at the cost of the symbolic formulas for each path becoming much larger and slower to reason about. Essentially this approach delegates more of the exploration to the decision procedure. It can improve performance overall because the decision procedure can use many of its own optimizations, though representing and reasoning about large formulas can increase memory usage.

The table lookup approach is perhaps more natural in source-level symbolic execution systems that know when a variable has an array type. In binary-level symbolic execution, the first challenge is to recognize when a table lookup is occurring. Our system detects a table lookup when the effective address of a load is the sum of a constant value and a symbolic one, when the constant value is in the range of a memory address, and the symbolic value (treated as the table index) is bounded. For loads, we use a power-of-two bound, matching the construction of the lookup formula. For stores we compute an exact upper bound, since imprecise stores would introduce spurious symbolic values that hurt performance. In some cases the bound on the index expression is evident from its syntax (for instance, if it is zero-extended from a byte value); if not, our tool uses additional decision procedure queries in a binary search to find the tightest bound. The maximum allowed table size is configurable; for this paper, it was 2^{16}.

IV. Using transformation structure

The previous section described how to speed up symbolic execution when applied to inverting a single transformation. This section describes a more fundamental improvement: performing symbolic execution separately for each layer of a multi-step transformation, by using dynamic or static analysis to recover information about the structure of transformations. First we describe the data structure we use to represent this information (Section IV-A), then how to recover structural information either dynamically from a binary (Section IV-B) or statically using source code (Section IV-C). Finally we describe our step-by-step approach to constructing pre-images (Section IV-D).

A. The HI-CFG

In previous work [5], [4] we proposed a program representation called a Hybrid Information- and Control-Flow Graph ("HI-CFG" for short, pronounced "high-C-F-G"). A HI-CFG includes information about control flow, as in a control-flow or call graph, with nodes that represent program code blocks and edges that represent the executes-after relationship between them. A HI-CFG also includes information about data: nodes that represent data structures, and edges that represent the flow of information between them. Last but not least a HI-CFG captures the connection between control and data with *producer* and *consumer* edges between data nodes and code nodes. Specifically a consumer edge runs from a data structure to code that reads it, and a producer edge runs from code to a data structure it writes. A more detailed definition of the HI-CFG, as well as examples, are found in our previous work [4].

For purposes of the present work, the key feature of the HI-CFG is that it contains just the right information for finding a sequence of transformations and the code that implements them. Specifically, some of the data-structure nodes in the HI-CFG represent parts of the program that hold the inputs to or outputs from transformations, with the effect of the transformation itself showing up as an information-flow edge from the input to the output. Given the identity of the code that exhibits interesting behavior (i.e., the function in which a vulnerability occurs), the source of the transformed data values that trigger the behavior will be one of the consumer edges of that code. By following a chain of information-flow edges backward to a program input, we can recover the chain of transformations the data underwent. The HI-CFG's producer-consumer and control-flow edges also highlight the code that implements each transformation. For a transformation from data-structure node A to data-structure node B, the closest call-graph ancestor of the consumers of A and the producers of B will be the function whose execution performs the transformation: this function will be our target for symbolic execution.

B. Dynamic Binary Analysis

In security applications, it is often necessary to analyze vulnerabilities in software for which we do not have source code. With such applications in mind, our first approach for inferring transformation structure in a HI-CFG is a dynamic analysis based on an instruction-level trace. The full description of this analysis appears in previous work [5], [4], but for completeness we summarize the key aspects here.

A key challenge for binary analysis is determining the boundaries of data structures: inherently a binary treats memory simply as a large unstructured array of bytes, so we must re-infer which bytes constitute a single data structure. We focus on contiguous structures, which usually correspond to source-level arrays (we previously used the term "buffer"). Our system uses a combination of two approaches: one that groups memory accesses for the same indexed instruction, and another that groups accesses that occur in a linear pattern of addresses over time. Together with these inferred groupings, our system also follows the hierarchical structure of memory at a coarser granularity, for instance tracking stack frames and dynamic allocations so that it is aware when data structures are released.

Our system infers information-flow edges using a taint analysis which uses a unique taint marker for each data structure node. It also tracks the effect of tainted data on branches at the function level to over-approximate implicit flows. It infers call and return edges primarily from `call` and `ret` instructions, but also retains a shadow stack to handle non-local exits and tracks jumps to addresses that were also call targets to handle tail-call optimizations.

We implement the system for instruction traces, generated by the BitBlaze [11] Tracecap tool, for Linux/x86 binaries. The analysis system is implemented primarily in Python, but links to the XED2 library (also used in Pin [12], [13]) for instruction decoding. The tool can operate on stripped binaries without symbol tables or debugging information, with the sole exception that its tracking of dynamic allocations requires the user to specify the address of `malloc` and related functions.

C. Static Source Analysis

If source code is available, then static analysis of that source is also an appealing possibility for recovering transformation structure information. Source-level type information provides a good starting point for the layout of data structures and producer/consumer relations, which we can refine (or correct, for a type-unsafe language like C) with standard points-to analysis techniques, particularly for dynamically-allocated structures. Static data-flow analysis also naturally provides information-flow edges, and a static intra-procedural CFG can be used to more precisely bound implicit flows. Finally static techniques for call graph analysis are also well known.

The key difference in usage between a dynamically and statically constructed HI-CFG is that a dynamic HI-CFG represents only those data structures and transformations that occurred on one or more sample executions, whereas a static HI-CFG contains all possible transformation chains. Creating such a sample execution adds another step when using a dynamic approach, though we expect that often an existing test suite would provide a suitable sample input. This step is not needed with a static HI-CFG, though it might be useful to provide tools uses with another mechanism for specifying a desired transformation sequence if desired.

In the future, we have plans to build a static analysis of the kind described above based on the LLVM framework [14], [15] and its Data Structure Analysis [16]. However the experiments in this paper use the previously-described dynamic analysis.

D. Sequential Pre-image Search

After recovering the sequence of chained transformations that produces a value, our approach then uses that structure to more efficiently search for program inputs that are pre-images of that combination of transformations. Instead of treating all the transformations as a single unit, our approach is to invert each transformation separately, starting with the final one and working back until we reach the program input. We compute pre-images for each separate transformation using symbolic execution and the enhancements described above in Section III.

The key reason this divide-and-conquer approach is valuable is that allows our system to better skirt some of the scalability limitations of symbolic execution. If there are m feasible execution paths through one transformation, and n feasible paths through a second transformation, and every combination of the two paths is feasible, then there will be mn paths for symbolic execution to explore through the composition of the two transformations. Usually some combinations of paths are not feasible, but we still observe that the number of feasible paths through composed transformations grows multiplicatively. By contrast, when computing pre-images for the two transformations separately, the effort required by the two separate symbolic execution searches grows additively, like $m + n$.

Our system currently implements a simple sequencing between the pre-image computation steps for each transformation. Suppose that the program has two transformations: F followed by G. Given a value z that triggers interesting behavior such as a crash, our system will first search for a pre-image y_1 such that $G(y_1) = z$. After succeeding there, it will then search for a further pre-image x_1 such that $F(x_1) = y_1$. This simple sequencing works well in the common case when the transformation F is surjective, but if it is not, there is a possibility the search can get stuck: it could compute a y_2 which $G(y_2) = z$, but for which there does not exist a x_2 with $F(x_2) = y_2$. In such a case the second search will fail. We discuss possible enhancements for dealing with this situation in Section VIII-A.

V. FuzzBALL implementation

FuzzBALL is a highly-configurable symbolic execution engine that builds on top of the BitBlaze Vine library [17], [11] to support binary applications. It has been developed (primarily by the first author) since 2009, and applied to a number of applications including static-guided test generation [18] and analysis of CPU emulators [19].

FuzzBALL takes the form of an emulator for standalone binary code or Linux user-mode programs, into which the user can introduce symbolic variables in place of the machine state. FuzzBALL will automatically explore the execution paths feasible by choosing values for the symbolic variables, and can check for user-specified logical conditions over the machine state as well as (relevant for this paper) the production of particular outputs. One notable simplification compared to systems such as KLEE is that FuzzBALL does not "fork" to explore paths in parallel; it instead runs one path to completion before starting another.

FuzzBALL consists of about 18,000 lines of code in the object-oriented/functional language OCaml (also used by Vine). Both programs code and symbolic expressions are represented in the "Vine IL" intermediate language, symbolic expressions in the purely-functional expression subset. OCaml's pattern matching features facilitate transformations and optimization of these representations. OCaml's compile-time "functor" polymorphism (somewhat similar to C++ templates) is used so that the same interpreter code can be instantiated in either concrete or symbolic versions. FuzzBALL translates binary code into the Vine IL on the fly, much like a dynamic binary translation system, so no static disassembly is required. Because binary code can access memory at varying granularity (e.g. first as a word and then as four bytes), FuzzBALL's symbolic memory representation uses the granularity of the last write, but split or reassemble values as needed. FuzzBALL generates bitvector constraints from symbolic branches and solves them by using an external decision procedure, currently STP [6] or Z3 [7].

The source code for FuzzBALL and the parts of the Vine library it uses are available under an open-source license [20].

VI. Pre-image technique evaluation

This section evaluates the performance of different configurations of FuzzBALL and KLEE for two different transformations (HEX decoding and RLE – Run Length Encoding – decoding) with different lengths of input and output.

We evaluate the following four different program configurations:

KLEE the most current version of KLEE [3] (r178863, released on Apr-05 2013) with the supplied uclibc and posix-runtime using the random-path search strategy described in [3]. In addition, we modified KLEE to terminate symbolic execution when the first match was found (instead of exploring the complete symbolic path). We use this configuration to compare FuzzBALL to other symbolic execution engines;

FuzzBALL the current version of FuzzBALL without transformation-guiding heuristics or sequential pre-image searching using HI-CFG information. This configuration is used to show performance of naive, unguided symbolic execution;

FuzzBALL-heuristic the current version of FuzzBALL with active transformation-guiding heuristics as introduced in Section III;

FuzzBALL-HI-CFG the most optimized version of FuzzBALL with all search optimizations and active usage of the HI-CFG to split sequences of transformations to compute sequential pre-images.

For these experiments we evaluate two different transformations, HEX decoding and RLE decoding as implemented and used, e.g., in PDF document viewers. HEX decoding takes an ASCII string that may only contain (i) an even number of the numbers 0-9 and letters a-f and A-F and (ii) any number of white-space characters (e.g., newline, tab, or space). Always letter/number characters are decoded into a byte in the output stream. RLE decoding describes a simple transformation where the first byte either describes the number of verbatim copied bytes or the number of repetitions of the next byte. An RLE encoded stream always ends with an 0x80 byte. PDF files may contain different (nested) objects with different encodings and a specific object might be RLE and HEX encoded, i.e., the data is first compressed using RLE and non-printable characters are then escaped using HEX encoding.

For our performance study we use two sample programs that use the before-mentioned transformations. RLE uses one simple RLE transformation (i.e., FuzzBALL-HI-CFG has the same performance as FuzzBALL-heuristics because there is only one transformation). The second experiment uses first a HEX transformation followed by a RLE transformation.

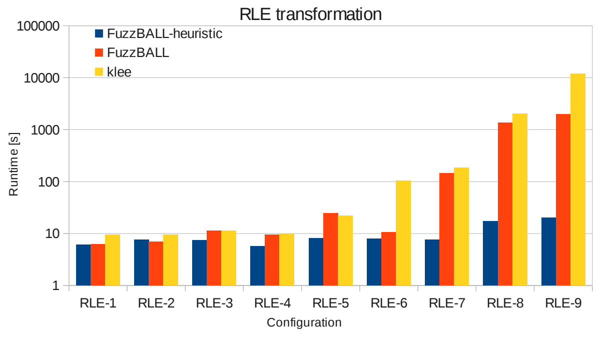

Fig. 1. Different RLE configurations for different symbolic execution configurations; runtime is in seconds on a logarithmic scale, lower is better.

We execute all benchmarks on an Intel Xeon CPU X5670 at 2.93GHz with 12 physical cores, 24 threads, and 32GB of main memory. Every individual benchmark configuration is executed 5 times and we report the average of these 5 runs.

A. RLE decoding

In this setting we evaluate a simple program that takes an input stream and RLE decodes the given input stream. We use KLEE and the different FuzzBALL configurations to evaluate the performance of symbolic execution. Table I shows the different configurations of the RLE decoding experiment. The input bytes are symbolic and symbolic execution computes a concrete version of the input for a given output.

TABLE I
LIST OF CONFIGURATIONS FOR THE RLE PROGRAM THAT USES ONE RLE DECODE TRANSFORMATION. SYMBOLIC INPUT (IN BYTES) IS TRANSFORMED TO AN OUTPUT STRING.

Configuration	Input [b]	Output [b]	Output
RLE-1	3	3	“AAA”
RLE-2	3	4	“AAAA”
RLE-3	3	6	“AAAAAA”
RLE-4	4	2	“AB”
RLE-5	5	3	“ABC”
RLE-6	5	6	“AAAAAB”
RLE-7	5	7	“AAAAAAB”
RLE-8	5	12	“AAAAAABBBBBB”
RLE-9	7	9	“AAAAAABCD”

Figure 1 shows the different configurations and the corresponding runtime. For KLEE the experiments show that the runtime primarily depends on the length of the input string (the number of symbolic input bytes) and secondarily the number of output bytes (the number of bytes that depend on the input string and are produced using a computation of the input bytes). With longer inputs we see an exponential increase of the runtime due to the state explosion with which the random-path search strategy cannot cope with. The results of FuzzBALL without heuristics are comparable to KLEE. FuzzBALL-heuristics on the other hand ranges from 5.1s to 15.0s and scales to larger input and output sizes as well.

B. HEX RLE decoding

In this setting we evaluate a program that uses two transformations that are chained (i.e., the input flows through one transformation and is then passed to a second transformation). Looking at the input the first transformation takes the input and HEX decodes a given input string. The HEX decoded values are then decompressed using RLE decoding. A symbolic execution engine that does not use the HI-CFG targets a specific output and reverses both the RLE and the HEX decoding as one complex transformation. The HI-CFG configuration splits this large task into two sequential transformations and first reverses the RLE decoding, effectively RLE encoding the given output. The second transformation of the HI-CFG configuration then HEX encodes the given RLE string from the first transformation.

Table II shows the different configurations for the HEXRLE binary. Monolithic symbolic execution engines (e.g., the KLEE and FuzzBALL-heuristics configuration) mark the input bytes symbolic and computes a pre-image for both transformations in one step, thereby directly targeting the output using the given symbolic input bytes. A sequential symbolic execution engine (i.e., FuzzBALL-HI-CFG) on the other hand splits the single large computation into a sequence of transformations and first uses an intermediate (Inter. in Table II) amount of symbolic bytes to first reverse the second transformation. Using the result from this step the first transformation is reversed using a given input of symbolic bytes.

TABLE II
LIST OF CONFIGURATIONS FOR THE HEXRLE PROGRAM THAT USES TWO TRANSFORMATIONS: INPUT IS FIRST HEX DECODED AND THEN RLE DECODED AND MATCHED AGAINST A GIVEN OUTPUT STRING. THE ORIGINAL INPUT IS MARKED SYMBOLIC.

Configuration	In. [b]	Inter. [b]	Out. [b]	Output
HEXRLE-1	10	5	12	“AAAAAABBBBBB”
HEXRLE-2	14	7	9	“AAAAAABCD”
HEXRLE-3	16	8	10	“AAAAAABCDE”
HEXRLE-4	18	9	11	“AAAAAABCDEF”
HEXRLE-5	125	60	57	“This is a longer test for symbolic execution and FuzzBall”
HEXRLE-6	250	120	114	twice HEXRLE-6

Figure 2 shows the different configurations for the symbolic execution engines and the corresponding runtimes on a logarithmic scale. With an increasing number of symbolic input bytes monolithic symbolic execution reaches its limit and both KLEE and FuzzBALL-heuristic run into timeouts for larger inputs.

The longer running experiments show that regular symbolic execution cannot cope with the large amount of possible states and KLEE was stopped after 10 hours of computation without producing any results. Even for FuzzBALL with heuristics the longer experiments were not feasible. Only when using a combination of sequential transformations and heuristics symbolic execution is able to reverse multiple transformations. FuzzBALL-HI-CFG on the other hand is still able to compute

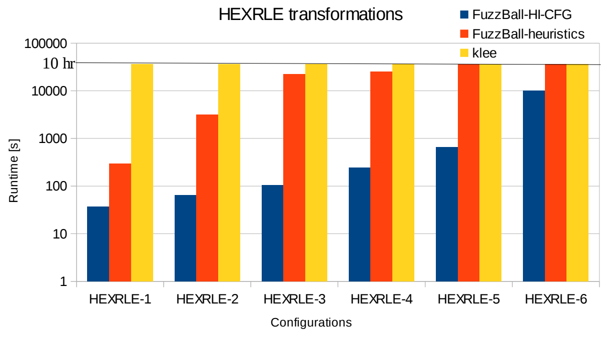

Fig. 2. Different HEXRLE configurations for different symbolic execution configurations; runtime is in seconds on a logarithmic scale, lower is better.

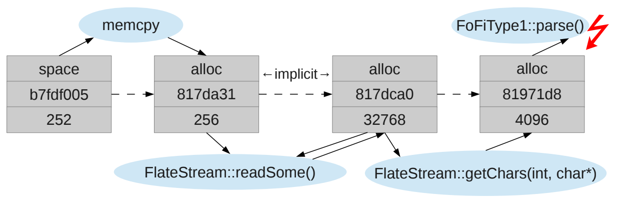

Fig. 3. Path to the vulnerable function in the HI-CFG.

pre-images for individual transformations and to efficiently reverse chained transformations with up to *250 symbolic bytes*.

VII. Exploit generation case studies

In this section we look at two applications (Poppler in Section VII-A and AbiWord in Section VII-B) with different security vulnerabilities. The vulnerabilities are hidden “deep” in the program and are only triggered after a set of transformations from the original input.

The security implications of these results are discussed in more detail in [4], which also includes performance measurements of the HI-CFG construction tool; here we focus on the transformation pre-image computation.

As a first step one can use FuzzBALL and a guess of a potentially vulnerable function to construct a “unit exploit”: a set of parameters passed to a function which cause it to crash in a dangerous way. We omit the details because they are not relevant to the present paper, but they can be found in the technical report [5]. Then the second step is to turn that unit test into a system test: a proof-of-concept exploit. Using the unit exploit and a HI-CFG generated from a benign input (one that does not trigger the bug), we use transformation-aware symbolic execution to build a program input that triggers the same dangerous crash as the unit exploit. This proves that vulnerability is a real problem, and can be used as a starting point for debugging and fixing it.

Both programs are open-source and we use the source-code to verify both the presence of the bugs and the correctness of our results. In the automatic analysis and HI-CFG construction our system does not use the source-code but uses only the binary itself to construct an exploitable input.

A. Poppler

Poppler is an open-source PDF library used in several applications, e.g., in viewers like Evince. In this case study we generate a vulnerability for CVE-2010-3704 [21] which allows arbitrary memory writes.

PDF documents may contain many different objects and one type of objects is used for embedded Type 1 fonts. Type 1 fonts are derived from PostScript and allow character descriptions in a flexible text format. Poppler includes a simple font parser that recovers the character encoding by mapping byte values to character glyphs. The standard allows unsigned integers between 0 and 255 inclusive. These integers may be specified as decimals or as octals with a prefix of `8#`.

The bug described in CVE-2010-3704 is an incorrect conversion from octal to decimal which does not check for negative numbers. The check in the code which ensures that values are smaller than 256 is a signed comparison and allows numbers smaller than 0. Malicious fonts may specify large integer values that when interpreted as signed values have the sign bit set and pass the range check. This vulnerability allows a malicious font to overwrite any 4-byte aligned memory location relative to the decompressed font object in memory. The unit exploit contains the string “`␣8#0027777774674`” (where the first character is a space) as a character index which trigger the vulnerability.

We use our system to create a proof of concept exploit for the described vulnerability by automatically constructing a compressed Type 1 font in a PDF that triggers the vulnerability.

In a first phase the HI-CFG of the Poppler library is constructed by executing `pdftoppm` (a sample program that uses Poppler to render PDFs as PPM images) with a benign PDF that contains a valid font. The benign PDF is generated with `pdftex` by rendering a small $\TeX$ file that contains a FlateDecode-compressed Type 1 version of the Computer Modern Roman 10 point font.

Figure 3 highlights the subset of the HI-CFG of Poppler that shows a path from the input to the vulnerable function. The input passes through a sequence of four transformations: the first two transformations are due to input buffering in C/C++ where data is copied from one buffer to the other; the third transformation is a flate decompression step where the Type 1 input font is decompressed; the forth step is again a copy from the decompression object to the Type 1 font object.

Using this HI-CFG information and the vulnerable condition the symbolic execution engine trivially reverses the fourth transformation and continues with the deflate compression. The symbolic execution engine then spends most of its time computing the pre-image for the third transformation. This non-trivial inversion computes a compressed font that decompresses to the attack font. The second and first transformation are again trivially inverted by just copying the data between the buffers. The symbolic exploration phase took less than 2 hours (6755.54s) on an Core 2 Duo E8400.

We also repeated the experiment with two other transformation chains (indicated with two other benign input files and the corresponding HI-CFGs).

Another commonly used transformation of streams in PDF files is RC4 encryption, used for password protection. Though some cryptographic functions are designed to be difficult to invert, RC4 stream encryption is this context is not difficult because RC4 is a stream cipher and the key is fixed: constructing a new ciphertext simply requires XORing the desired plaintext with the keystream. With no branching required, this is a nearly ideal case for symbolic execution: only one symbolic path needs to be explored, requiring 20 seconds mostly devoted to program startup.

Two further transformations supported in PDF files are the run-length encoding and hexadecimal encoding introduced in Section VI: in fact we were inspired to use the transformations for that experiment after seeing them in Poppler. Poppler's implementation is different from the one we used in Section VI, but the performance of transformation-aware symbolic execution is similar: the preimage computation requires 143 seconds and 315 symbolic paths.

B. *AbiWord*

AbiWord is a word processor that is able to import many different document formats. In this case study we examine the Office Open XML input filter that is used to parse documents from recent versions of Microsoft Word. Office Open XML documents are structured as a compressed Zip file that contain multiple XML documents which in turn represent the document contents and metadata.

Recent versions of AbiWord (we use 2.8.2) suffer from a crash in the XML processing library that is triggered when a `shading` tag occurs outside of a `paragraph` tag. The code tries to fetch the top element of a C++ STL stack which contains pointers to enclosing document objects. When the shading tag occurs outside of the expected areas this stack is empty and an invalid pointer is dereferenced. We have not yet determined if this crash is exploitable.

The generated HI-CFG contains 5139 functions and 7816 groups. Looking at the sequence of buffers in the HI-CFG, the document data starts in a standard-IO input buffer, and is then decompressed by the `inflate` function. The decompressed buffer is then copied unchanged via `memmove` into a structure called the parser context, which is used by `xmlParseDocument`; the function containing the vulnerability is a callback from this parser.

This vulnerability is located behind the Zip decompression transformation (which uses a related algorithm to the Deflate compression in PDF, though with an independent implementation). In our experiment the pre-image computation happened to run faster than in the Poppler example because we used a benign input file with a smaller Huffman tree and smaller decoding tables. On average the pre-image search required 237 seconds and 92 symbolic paths.

VIII. Discussion and future work

In this section we further discuss some of the observed and predicted limitations of our current approach and implementation, and how they might be addressed in the future.

A. *Non-Surjective Transformations*

Our current sequential backward processing through transformations (Section IV-D) takes advantage of the assumption that each transformation will be surjective, so it suffices to generate a single pre-image. If applied with non-surjective transformations, our approach could get stuck by producing a pre-image from one transformation that is not a possible image of the preceding transformation. For instance, consider inverting the transformation of Latin-1 to UTF-8 conversion. This transformation is not surjective: specifically only byte values 0x00 through 0xc3 will appear in the UTF-8 encoding of Latin-1 bytes. Thus if the inversion of a following transformation produced a preimage containing 0xdd, the search for a corresponding Latin-1 string would be guaranteed to fail. (The Latin-1 to UTF-8 encoding process is simple enough that the search would fail quickly, but for a more complex transformation the fruitless search might be very long.)

Two general search techniques that might be applied to deal with this challenge would be to backtrack from a failed search and go back to search for another pre-image from a previous transformation; or to generate a set of multiple pre-images at each step. However these techniques would only be effective if the chance that an arbitrarily selected value is an image of a transformation is large, which is not guaranteed in general. For instance if one generates a 76-byte string uniformly at random, the chances that it contains no bytes larger than 0xc3 are about one in a billion. A more sophisticated approach would be to infer a regular language or context-free grammar that captured (some of) the constraints on a transformation output (compare [22]), and then to add this grammar as a constraint on the symbolic execution process [23].

B. *Transformation-level Path Explosion*

Another potential challenge related to the high-level structure of our search across multiple transformations is that there might be a large number of possible sequences of transformations that connect the program input to a vulnerable or otherwise interesting function. If the user of our tools is interested in only one such path, they can specify it, for instance via the choice of test inputs for a dynamic analysis. Also we expect that in many cases, many of the possible sequences of transformations would be equally effective in building a system test, as for instance in our Poppler case study. But one could imagine a case in which many sequences of transformations appear possible at the level of the HI-CFG, while only a small fraction of these are actually feasible to create a system test for a particular behavior. In such a case, the search for the correct chain of transformations could be in a large space, so we would techniques to guide it as well.

C. *Recognizing Transformation Outputs*

Not all transformations write their outputs to a contiguous array (as currently assumed by our symbolic execution strategies) in sequential order (as recognized by one part of our dynamic data-structure analysis). One natural direction for generalizing our approach would be to recognize more styles of transformation output. More sophisticated data-structure analysis could help generalize our system to linked data structures such as lists and trees. Another common pattern in streaming transformations (which we observed for instance in Poppler) is for each output value to appear as the return value of a function call. One convenient aspect of this pattern for our techniques is that automatically enforces that the output is sequential.

D. *Multi-threaded Subject Programs*

One of the current practical limitations of FuzzBALL is that it does not support multi-threaded subject programs. For instance this is the most fundamental obstacle keeping us from extending the Poppler results to Adobe Reader (admittedly Reader is also noticeably larger). However such support is independent of the approaches in this paper, and our dynamic analysis has already been designed with multi-threaded programs in mind.

IX. Related work

We concentrate here on related approaches to inversion and symbolic execution. A comparison with related program representations [24], [25], exploit generation techniques [26], [27], and binary structure inference systems [28], [29], [30], [31] can be found in the companion technical report [5].

The "decomposition and restitching" work of Caballero et al. [32] also tackles the problem of creating a transformed program input that triggers a vulnerability (there in malware), but they focused on transformations such as decryption functions which cannot be effectively be inverted by symbolic execution: instead they searched for inverse functions within a binary. The Inspector Gadget system [33] computes pre-images of transformation functionality extracted from a binary using a technique called "gadget inversion". Gadget inversion uses only concrete executions, but it records input-output dependencies which can be seen as a subset of symbolic execution. Our approach of working backwards through a program's information flow is analogous to previous program analysis approaches that worked backwards through control-flow [34], [35] to trigger program behavior. Instead of computing a single pre-image, symbolic execution techniques can also be used as part of synthesizing a complete inverse program [36]. Though more general when possible, this is currently less scalable: it applies to compression algorithms similar to Deflate, but expressed as 25 lines of high-level code, and it requires a human assistance in producing a template.

MAYHEM [10] is a binary symbolic execution system based on a library and techniques similar to Vine and FuzzBALL, and aimed at exploit generation. In particular MAYHEM introduces a index-based memory model which provides similar benefits to our treatment of symbolic memory accesses in Section III-D (developed independently). Though MAYHEM is demonstrated discovering a number of exploits, none appear to involve input transformations as we concentrate on here.

X. Conclusion

Generating a system test from a unit test is difficult, in part because program inputs may undergo complex transformations before reaching a unit that has interesting behavior (such as a bug or security vulnerability). To tackle this challenge we introduce transformation-aware symbolic execution techniques, which optimize the search for transformation pre-images and apply a divide-and-conquer approach based on the inferred composition structure of transformations. We implement these new techniques in a sophisticated binary-level symbolic execution system FuzzBALL. In a direct comparison, these enhancements improved the performance of our symbolic execution tool above both the baseline tool and an independent state-of-the-art system, often by multiple orders of magnitude. Applied in a case study of vulnerable document processing applications, our system generated crashing program inputs from unit crash information, using only a binary program without source code.

Acknowledgments

We thank László Szekeres and Lenx Tao Wei for suggestions and help related to the experiments and previous papers. This work was supported by NSF awards CCF-0424422, 0842695, and 0831501; MURI awards N000140911081 (ONR) and FA9550-09-1-0539 (AFOSR), and DARPA award HR0011-12-2-005. However the findings and conclusions are those of the authors and do not necessarily reflect the views of the NSF or other supporters.

References

[1] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for Java," in *ASE'05*.
[2] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *FSE'06*.
[3] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*.
[4] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song, "HI-CFG: Construction by binary analysis, and application to attack polymorphism," in *ESORICS'13*.
[5] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song, "Transformation-aware exploit generation using a HI-CFG," University of California, Berkeley, Tech. Rep. UCB/EECS-2013-85, May 2013.
[6] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV'07*.
[7] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS'08*.
[8] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE S&P'10*.
[9] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS'11*.
[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing MAYHEM on binary code," in *IEEE S&P'12*.
[11] "BitBlaze: Binary analysis for computer security," http://bitblaze.cs.berkeley.edu/.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI'05*.

[13] Intel, "Pin website," *http://www.pintool.org/*, Nov. 2012.

[14] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, USA, Mar. 2004, pp. 75–88.

[15] "The LLVM compiler infrastructure," http://www.llvm.org.

[16] C. Lattner and V. Adve, "Data structure analysis: An efficient context-sensitive heap analysis," Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2003-2340, Apr 2003.

[17] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *ICISS'08, keynote invited paper*.

[18] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *ISSTA'11*.

[19] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *ASPLOS'12*.

[20] "FuzzBALL binary symbolic execution tool," http://bitblaze.cs.berkeley.edu/fuzzball.html.

[21] MITRE, "CVE-2010-3704: Memory corruption in FoFiType1::parse," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3704, Oct. 2010.

[22] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery," in *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[23] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI'08*.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *TOPLAS'87*, vol. 9, no. 3.

[25] S. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *TOPLAS'90*, vol. 12, no. 1.

[26] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *IEEE S&P'10*.

[27] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *NDSS'11*.

[28] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *NDSS'10*.

[29] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a dynamic excavator for reverse engineering data structures," in *NDSS'11*.

[30] ——, "Body armor for binaries: preventing buffer overflows without recompilation," in *USENIX ATC'12*.

[31] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *NDSS'11*.

[32] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *CCS'10*.

[33] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries," in *IEEE S&P'10*.

[34] Y. Xie, A. Chou, and D. R. Engler, "ARCHER: using symbolic, path-sensitive analysis to detect memory access errors," in *ESEC/FSE'03*.

[35] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *SAS'11*.

[36] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, "Path-based inductive synthesis for program inversion," in *PLDI'11*.