Execution Hijacking: Improving Dynamic Analysis by Flying off Course

Petar Tsankov, Wei Jin, Alessandro Orso
Georgia Institute of Technology
Email: {peshko|weijin|orso}@cc.gatech.edu

Saurabh Sinha
IBM Research – India
Email: saurabhsinha@in.ibm.com

***Abstract*—Typically, dynamic-analysis techniques operate on a small subset of all possible program behaviors, which limits their effectiveness and the representativeness of the computed results. To address this issue, a new paradigm is emerging: execution hijacking, consisting of techniques that explore a larger set of program behaviors by forcing executions along specific paths. Although hijacked executions are infeasible for the given inputs, they can still produce feasible behaviors that could be observed under other inputs. In such cases, execution hijacking can improve the effectiveness of dynamic analysis without requiring the (expensive) generation of additional inputs. To evaluate the usefulness of execution hijacking, we defined, implemented, and evaluated several variants of it. Specifically, we performed an empirical study where we assessed whether execution hijacking could improve the effectiveness of a common dynamic analysis: memory error detection. The results of the study show that execution hijacking, if suitably performed, can indeed improve dynamic analysis.**

"*When you come to a fork in the road, take it.*"
–Yogi Berra[1]

I. Introduction

Dynamic-analysis techniques, which instrument a program and monitor its behavior during execution, are commonly used for many applications, such as fault detection (*e.g.,* [1]), memory error detection (*e.g.,* [2], [3]), code optimization (*e.g.,* [4]), regression testing (*e.g.,* [5]), invariant detection (*e.g.,* [6]), specification mining (*e.g.,* [7]), and classification of program behaviors (*e.g.,* [8]). On the one hand, such techniques overcome imprecision, which is one of the main limitations of static analysis—static-analysis techniques typically compute inaccurate results because they operate on an over-approximation of a program's behavior, by either traversing infeasible program paths or making conservative approximations of properties that are difficult to reason about statically (*e.g.,* heap structures).

On the other hand, dynamic-analysis techniques are intrinsically incomplete, as they operate on an under-approximation of a program's behavior; the information they compute is only as good as the comprehensiveness of the set of behaviors the analysis can observe. Because typical workloads (*i.e.,* program inputs) used to perform dynamic analysis explore only a small fraction of potential program behaviors, the analysis results computed using such workloads can be incomplete to a large extent. For example, a dynamic memory-leak detection technique may miss many leaks if they are not exercised by the inputs considered.

To alleviate this limitation of dynamic analysis, researchers have investigated ways to improve the representativeness of developer-provided program inputs by leveraging field executions (*e.g.,* [1], [8]–[11]) and observing program behaviors much more extensively than what is possible in-house. Performance and privacy concerns, however, limit the applicability of such techniques in practice.

Researchers have also developed many approaches for providing automated support for input generation. Especially popular among those are approaches based on symbolic execution (*e.g.,* [12]–[17]), which generate inputs by interpreting a program over symbolic values and solving constraints that lead to the execution of a specific program path. Despite the availability of increasingly efficient techniques, decision procedures, and machines, however, symbolic execution is still limited by both scalability and practical applicability issues (*e.g.,* the combinatorial explosion of the number of paths, the presence of libraries, and the interactions between the program and the external environment).

More recently, some researchers have started investigating techniques that force executions along specific paths in an unsound way, so as to explore a larger set of program behaviors [18]–[20]. Instead of trying to generate better or larger input sets, these techniques aim to augment the set of behaviors covered by an existing set of inputs. We call these approaches *execution hijacking* because, intuitively, they involve taking control over an execution to drive it towards unexplored behaviors. More specifically, execution hijacking monitors a program execution and forces control to proceed along otherwise unexplored parts of the program. To do this, the approach typically flips the outcome (true or false) of one or more predicates, so that the path of execution follows a branch that has not been traversed before (and that would not have been traversed in the original execution).

The advantage of execution hijacking is that, given a program P and a set of inputs I for P, it allows for exercising a potentially much larger number of behaviors of P than a traditional execution of P against I. The obvious drawback of the approach is that it introduces unsoundness in the execution: a hijacked execution of P with input i traverses program paths that would not be traversed by the normal execution of P with i. For example, the hijacked execution may cause a runtime exception that would typically not

[1]When giving directions to his New Jersey home, which was equally accessible via two different routes.

occur in the normal execution and may be a false positive. However, there are cases where the manifested behavior could occur for some other normal execution with a different input i'. In these cases, the hijacked execution can expose such behavior without requiring the generation of input i'. In general, the usefulness of the technique depends on how often a behavior manifested by a hijacked execution is infeasible.

In this paper, we perform an investigation of execution hijacking and a study of its ability to improve dynamic analysis. To this end, we start by providing a general definition of the approach and discussing several variants of it. These variants include different ways to mitigate the infeasibility problem—some adapted from related work and some new. We then introduce our prototype tool, NIONKA, that is parametrized so that it can run our different variants of the technique. Finally, we present an empirical study in which we assess the usefulness of execution hijacking by using NIONKA to improve the effectiveness of a commonly performed dynamic analysis: memory error detection [21].

More precisely, in our empirical evaluation we used NIONKA to perform execution hijacking on sets of inputs of different sizes for several applications while performing memory error detection. We then compared the results of the analyses performed with and without NIONKA and measured the effectiveness of execution hijacking in improving such results. Although our investigation is still at its early stages, our results are promising and motivate further research. For the programs and input sets considered, memory error detection combined with execution hijacking was always able to identify more errors than memory error detection alone. Moreover, the analysis of the performance of the different variants considered provides some insight on successful ways to mitigate the infeasibility problem and, ultimately, reduce the number of false positives.

If our initial results were confirmed by further studies, execution hijacking could represent a considerable step forward for dynamic analysis because it (1) can be applied to programs of any complexity that take inputs of any complexity, (2) requires no expensive analysis, and (3) is easily implementable through lightweight instrumentation. The main contributions of this work are:

- A general definition of execution hijacking, an approach for increasing the number of behaviors observed by dynamic analysis without generating new inputs.
- Several variants of the basic approach that can help mitigate the infeasibility problem that is inherent in execution hijacking.
- The development of NIONKA, a tool that implements our different variants of execution hijacking and that is freely available.[2]

[2]The tool can be downloaded at http://www.cc.gatech.edu/~orso/nionka/

```
    Inputs: {<100, 100>, <0, 0>, <-100, -100>}
```

```
void main(int x, y) {
1. z = complex(x);
2. if (z > 0)
3.   print (100 / y);
4. else
5.   print (100);
}
          (a)
```

```
void main(int x, y) {
1. z = complex(x);
2. if ((z > 0) && (y != 0))
3.   print (100 / y);
4. else
5.   print (100);
}
          (b)
```

Figure 1. Examples of successful and unsuccessful applications of execution hijacking.

- An initial study that shows feasibility and potential usefulness of execution hijacking and the characteristics of the approach that can help making it more effective.

II. Execution Hijacking

In this section, we first introduce the concept of execution hijacking using an example and then discuss it in detail.

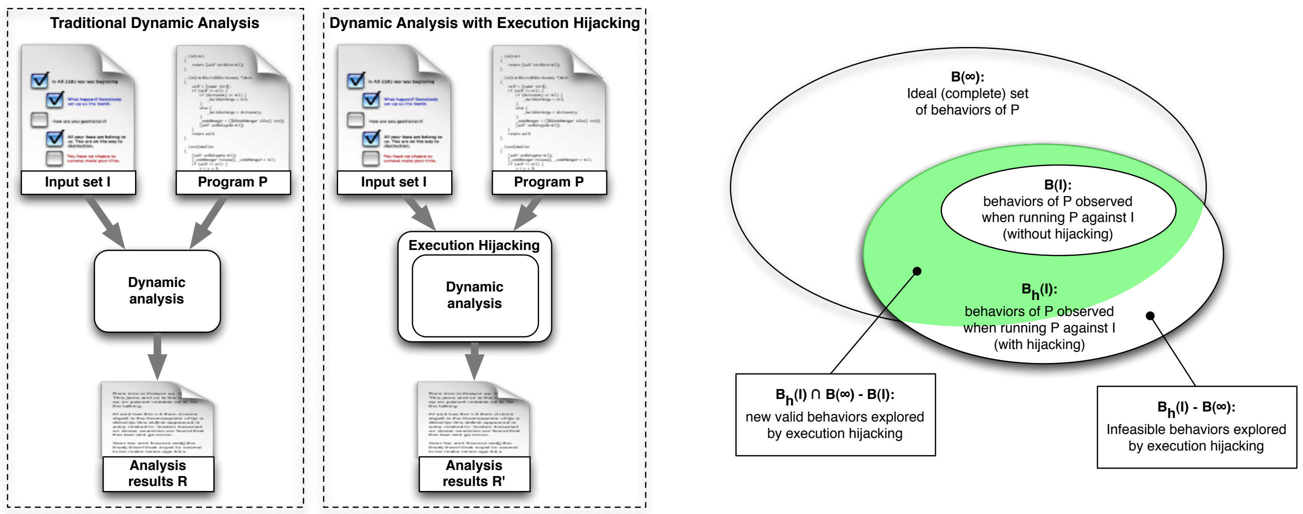
A. Overview

The basic idea behind execution hijacking is straightforward: we want to forcibly exercise program behaviors that would not otherwise be exercised by a given set of inputs. To do this, the approach forces an execution to follow a specific path by flipping the outcome of some predicates. To illustrate the basic approach, we use the two small code examples shown in Figure 1.

The two examples are almost identical; the only difference between the two programs is the predicate at line 2. Consider a scenario where the dynamic analysis we are interested in is some form of smoke test [22]: that is, we execute the application against a set of inputs and check whether it crashes or runs to completion. Assume that the inputs we are using are the ones shown in the figure (same inputs for both programs). Assume also that none of the values used for `x` causes the return value of `complex`, and thus `z`, to be greater than zero.

For the example in Figure 1(a), execution hijacking could be used to flip the outcome of the predicate at line 2, so as to cover that yet uncovered part of the program. By doing so, the execution of the second input would result in a failure due to a division by zero at line 3. In this case, although the hijacked execution is infeasible for input `<0, 0>`, it still identifies an actual problem with the code that could occur for any input `<x, y>` where (1) `y` is zero and (2) the value of `x` causes `complex` to return a positive value. Most importantly, it does so without having to identify such value of `x` and simply using the existing set of inputs for a program (*e.g.,* the program's test suite).

For the example in Figure 1(b) too, execution hijacking could be used to flip the outcome of the predicate at line 2. And also in this case, for the second input, the flipping would result in a failure due to the division by zero at line 3. In this case, however, the problem identified is a false positive—the predicate that we flipped is checking exactly that `y` has a non-zero value.



Traditional Dynamic Analysis
Input set I
Program P
Dynamic analysis
Analysis results R
Dynamic Analysis with Execution Hijacking
Input set I
Program P
Execution Hijacking
Dynamic analysis
Analysis results R'
B(∞): Ideal (complete) set of behaviors of P
B(I): behaviors of P observed when running P against I (without hijacking)
B_h(I): behaviors of P observed when running P against I (with hijacking)
B_h(I) ∩ B(∞) - B(I): new valid behaviors explored by execution hijacking
B_h(I) - B(∞): Infeasible behaviors explored by execution hijacking

Figure 2. Intuitive view of execution hijacking, its benefits, and its drawbacks.

These two examples illustrate the best and worst cases for execution hijacking. In the former case, execution hijacking reveals a feasible behavior and improves the effectiveness of dynamic analysis. In the latter case, conversely, execution hijacking generates an infeasible behavior that results in a false positive in the analysis. The relative frequency of the two cases, which we investigate in Section IV, determines the usefulness of the approach in practice.

B. Terminology and Definitions

In this section, we provide the terminology and definitions that we use in the rest of the paper.

Given a program P and an input i, we indicate with $e_n(P, i)$ the normal (*i.e.,* non-hijacked) execution of P with i as input. Given a program P and a set of inputs I, we call $\mathcal{E}_n(P, I) = \{e(P, i) \mid i \in I\}$ the set of normal executions of P when run against all of the inputs in I.

Given a program P, an input i, and a predicate s, we call $e_h(P, i, s)$ the hijacked execution of P that is obtained by running P against i and flipping the outcome of predicate s when it is evaluated. Similarly, given a program P, an input i, and a set of predicates S, we call $\mathcal{E}_h(P, i, S) = \{e_h(P, i, s) \mid s \in S\}$ the set of hijacked executions of P obtained by running P against i once for each predicate $s \in S$, and flipping the outcome of s during that execution. Finally, given a program P, a set of inputs I of cardinality n, and a set $\mathcal{S}$ of sets of predicates $\{S_1, ..., S_n\}$, one for each $i \in I$, $\mathcal{E}_h(P, I, \mathcal{S}) = \{\mathcal{E}_h(P, i_j, S_k) \mid i_j \in I, S_k \in \mathcal{S}, \wedge\, j = k\}$.

Given a program P and the execution of the program for a given input i, there are many places where the execution can be hijacked. Basically, each predicate in the program that is reached by i can be flipped to force the program down a different path. However, it makes little sense to force the execution along branches that would be exercised anyway. We therefore consider a predicate s as *flippable* with respect to an input i if s is covered by i, but one of the outgoing branches of s is not covered by the input. From a practical standpoint, the set $fp(P, i)$ of flippable predicates for a program P and a given input i can be easily computed by collecting branch coverage information for $e_n(P, i)$.

In this paper, we use the term *behavior* to indicate, intuitively, any observable aspect of an execution—which behaviors are of actual interest depends on the kind of dynamic analysis performed. If the analysis of interest identifies program failures, for instance, as in the two examples of Figure 1, any abnormal termination is a behavior of interest. Alternatively, if we are performing memory-leak detection, any memory allocation, deallocation, or access would be a behavior of interest. Although it may be possible to tailor execution hijacking to the specific dynamic analysis of interest, in its basic formulation, the approach is analysis-agnostic and simply tries to exercise as many behaviors as possible by steering the execution towards unexplored parts of the program.

Let b be a behavior of interest that is observed at a program point p, under state Γ, in a hijacked execution. The *relevant state*, $\Gamma_{rel} \subset \Gamma$, for b is the set of variables and associated values that characterize b at p. The *relevant state variables*, $vars(\Gamma_{rel})$, are the variables (or memory locations in general) over which state Γ_{rel} is defined. In the code of Figure 1, for instance, for the behavior in which a division by zero occurs at line 3, $\Gamma_{rel} = \{\langle \texttt{y} = 0 \rangle\}$, as the value zero for `y` leads to the observed behavior; and $vars(\Gamma_{rel}) = \{\texttt{y}\}$. To illustrate with another example, consider a behavior in which a null-pointer exception occurs at a statement that accesses a field `fi` of an object `obj` (*e.g.,* `y = obj.fi`); in such a case, $\Gamma_{rel} = \{\langle \texttt{obj} = \mathit{null} \rangle\}$, and $vars(\Gamma_{rel}) = \{\texttt{obj}\}$.

C. *Exploring Additional Behaviors to Improve Dynamic Analyses*

The two examples discussed in Section II-A illustrate how execution hijacking can explore additional program behaviors without requiring the use of new inputs. More generally, execution hijacking works by pushing the program into an inconsistent state in the hope of revealing interesting behaviors that are feasible despite the inconsistency.

Figure 2 provides an intuitive view of execution hijacking, its benefits, and its drawbacks. The left-hand side of the figure shows a typical usage scenario of execution hijacking. In the scenario, we have a program P, a set of inputs I for the program, and a dynamic analysis that would normally be performed on the set of executions $\mathcal{E}_n(P, I)$. When used in conjunction with execution hijacking, the dynamic analysis operates on the larger set of executions consisting of $\mathcal{E}_n(P, I)$ plus $\mathcal{E}_h(P, I, \mathcal{S})$, where $\mathcal{S}$ is the set of sets of branches to be flipped, one for each input $i \in I$, as defined in Section II-B.

The right-hand side of the figure shows the effect of execution hijacking on the set of behaviors observed by the dynamic analysis at hand. Set $B(\infty)$ indicates the ideal set of all possible behaviors of P; that is, the set of behaviors that would be exposed by exhaustively exercising the program with all possible inputs in the program domain. A dynamic analysis performed on this set of behaviors would be complete.

Set $B(I) \subset B(\infty)$ is the set of behaviors of P observed when P is run against all inputs in I. Typically, $B(I)$ is a very small subset of $B(\infty)$, and dynamic analyses performed on $B(I)$ produce results that are, therefore, incomplete. Execution hijacking, by forcing executions along unexplored parts of the program, exposes a set of behaviors $B_h(I)$ that can be divided into three categories: (1) behaviors that are also revealed by I $(B(I) \cap B_h(I))$, which do not provide any additional information to the dynamic analysis; (2) behaviors that are not revealed by I and are infeasible $(B_h(I) - B(\infty))$, which can cause the analysis to generate false positives; (3) behaviors that are not revealed by I and are feasible $(B_h(I) \cap B(\infty) - B(I))$, which would increase the set of behaviors that the dynamic analysis can observe and can, thus, improve the results of the analysis.

More precisely, let us consider a behavior b of interest that occurs, for a hijacked execution, at a program point p, with state Γ and relevant state Γ_{rel}. Behavior b is a valid behavior if there exists a normal execution e such that (1) p is reached in e with state Γ', and (2) $\Gamma_{rel} \subset \Gamma'$. In such cases, the behavior observed in the hijacked execution can also occur in a normal execution. In all other cases, b is infeasible and any analysis result derived from b is likely to be a false positive.

III. Instantiation and Implementation of the Approach

There are many ways in which execution hijacking can be instantiated, based on how the different aspects of the technique are defined. In our investigation, we instantiated and implemented several variants of execution hijacking, which allowed us to explore some fundamental characteristics of the general approach. In this section, we discuss these variants along two main dimensions. The first dimension relates to the extent of deviation from normal executions, that is, the extent or aggressiveness of the hijacking we implemented. The second dimension involves infeasibility-mitigation techniques that we defined or integrated into our variants. Note that these two dimensions are not completely orthogonal. For example, a larger deviation from normal executions may result in more infeasible results and thus require stronger mitigation techniques.

A. *Execution Hijacking Aggressiveness*

Given the set of flippable predicates for a program P and a set of inputs I, $\mathit{fp}(P, I)$, there are three main aspects that affect the aggressiveness of execution hijacking: which specific branches to flip, how many branches to flip in a single execution, and when to flip such branches.

Choice of the branches to flip: One straightforward possibility, when deciding which branches to flip for an input $i \in I$, is to go exhaustively and perform all of the executions in $\mathcal{E}_h(P, i, \mathit{fp}(P, I))$, that is, flip all branches in $\mathit{fp}(P, I)$. Another option is to use a random approach in which the branches to flip are chosen based on some probability distribution. Yet another possibility would be to use a directed approach that may pick the specific branch to flip based on some goal. For example, if we are using execution hijacking to support a dynamic analysis that targets specific program entities (*e.g.,* pointer dereferences), we may flip branches that are more likely to exercise one or more of such entities. In our study, we used an exhaustive approach that flips all flippable predicates because it gives us more data points for our investigation and more opportunities of studying the proposed infeasibility-mitigation techniques.

Number of branches to flip in a single execution: In general, the higher the number of predicates flipped in a single execution, the more inconsistent the program state and the lower the likelihood for the generated behaviors to be feasible. Therefore, we decided to flip at most one predicate per execution. In this way, we can better investigate the effectiveness of execution hijacking by limiting the number of factors involved in the study and the noise in the results.

Choice of when to flip a branch: Related to the issue of the number of predicates to flip in a single execution is the issue of when to flip them. Also in this case, there are various possibilities. A predicate may be flipped each time it is evaluated, the first time only, each other time, randomly, and so on. Our current implementation is generic enough that it can support all such options. In the study presented in Section IV, however, we decided to flip a predicate when it is evaluated the first time during the execution only, so as to reduce state deviation introduced by execution hijacking.

B. Infeasibility-mitigation Techniques

Because the usefulness of execution hijacking depends on how often a behavior manifested by a hijacked execution is infeasible and results in a false positive, it is important to try to limit the occurrences of such infeasible behaviors. To this end, we defined and adapted several infeasibility-mitigation techniques, which we describe in the rest of this section.

1) Filtering: Filtering removes from consideration apriori, before hijacking, those branches whose forced execution would likely result in infeasible behaviors. Our approach performs filtering based on code patterns, where a *code pattern* is a predicate with some specific characteristics. Currently, we use one pattern, the null-check pattern, which matches predicates that check whether a reference has a NULL value. Intuitively, these predicates are used to guard the access to the reference being checked, and flipping them is likely to result in a memory access error that could not occur in any normal execution. This is also confirmed by our experience with execution hijacking.

2) Early termination of hijacked executions: This technique is based on the intuition that more feasible behaviors are likely be revealed soon after hijacking occurs, before the corrupted state propagates widely. In other words, the longer an execution continues after hijacking, more infeasible behaviors are likely to be revealed because state corruption would become more pervasive. The technique, therefore, terminates a hijacked execution based on a stopping criterion. In our current formulation, we use coverage as a stopping criterion: our technique terminates a hijacked execution when the execution reaches an already covered statement. This is based on the rationale that additional behaviors are more likely to be revealed when traversing unexplored parts of a program. Although we did not consider additional criteria in this initial investigation, we plan to consider them in future work. Particularly, we will consider criteria based on different types of code coverage, such as data flow or path-based coverage.

3) State fixing: State fixing attempts to reduce state corruption by partially fixing the state after hijacking. Specifically, instead of forcing the execution of branch b at predicate p, the technique modifies the state of the program when p is reached so as to actually change the value of the predicate and cause b to be taken. For instance, if the condition of p is $x > 1$, p is supposed to be flipped, and $x > 1$ currently evaluates to true, the technique would change the state (in this case, the value of x) so that $x > 1$ evaluates to false instead (*e.g.,* by assigning a value of 0 to x). The rationale for state fixing is that operating on a state that is still corrupted, but at least locally in sync with the control flow, may reduce infeasibility. Unfortunately, fixing the state is a not-trivial operation in general because (1) predicates can be arbitrarily complex and involve constructs such as pointers and data structures, and (2) there are typically several ways to modify the state to change the outcome of a predicate [18]. To avoid these complications while still being able to investigate the effect of state fixing, our approach is currently limited to performing state fixing for predicates that involve scalar variables only.

C. The NIONKA *tool*

To empirically evaluate the variations of execution hijacking described in the previous section, we implemented them in a prototype tool called NIONKA. NIONKA can perform execution hijacking for C and C++, is built on top of the LLVM compiler infrastructure (http://llvm.org/), and leverages both the static analysis and the instrumentation capabilities of LLVM. One of the advantages of using LLVM is that it can generate target executable code for different architectures, which makes NIONKA more portable.

NIONKA performs predicate flipping for a program P and a set of inputs I as follows. First, it analyzes P to generate a list PR of all predicates in P and assigns a unique ID to each predicate. Then, NIONKA leverages LLVM's analysis capabilities to identify the predicates to be filtered out, as described in Section III-B1, and removes them from PR. In the next step, NIONKA collects coverage information for P when run against each input i_k in I and creates a set of lists B_{i_k}, where each list B_{i_k} contains the branches b_j, expressed as pairs $\langle predicateID, outcome \rangle$, such that input i_k executes b_j's predicate but does not execute b_j.

Finally, NIONKA replaces each predicate in PR with an alternative predicate that invokes a checking function chk and passes to chk (1) the outcome of the condition in the original predicate and (2) the predicate's ID. At runtime, when P is run against input i_k, chk checks whether a predicate should be flipped based on information read at the beginning of the execution from a configuration file containing the list B_{i_k} associated with input i_k. A parameter to NIONKA specifies how many times the branch should be flipped. For our experiments, we decided to flip only the first time the predicate is executed, as described in Section III-A.

Note that, although we have not leveraged this aspect of NIONKA in our evaluation, another parameter to the tool provides the possibility to change the flipping information at runtime, which enables more sophisticated flipping schemes. In fact, we implemented in NIONKA some of these more elaborated flipping schemes (*e.g.,* flipping only n times or only when a specific condition is satisfied).

IV. EMPIRICAL EVALUATION

To investigate both the general usefulness of execution hijacking in supporting dynamic analysis and the effectiveness of the different variations of the technique that

Table I
INFORMATION ON THE PROGRAMS USED FOR THE STUDY.

Program	Description	KLOC	# Tests	Coverage
flex	lexical analyzer	10	525	50%
grep	pattern-matching utility	10	800	33%
sed	stream editor	14	360	24%

we considered, we performed an empirical study. In the study, we used NIONKA to investigate the following research questions:

- **RQ1:** Can execution hijacking increase the effectiveness of dynamic-analysis techniques?
- **RQ2:** Can our infeasibility-mitigation techniques reduce the number of infeasible behavior generated by execution hijacking while still providing benefits?

In the rest of this section, we describe the experiment setup we used to investigate these questions and our results.

A. Experiment Setup

In this section, we describe the dynamic analysis that we targeted, the programs and data we used in our study, and our experiment protocol.

1) Dynamic analysis targeted: To investigate our research questions in a concrete context, we first selected a specific dynamic-analysis technique to target in the study. To make sure that our results are relevant, we selected a widely used dynamic analysis: memory error detection. And as a representative of memory error detection techniques, we chose MEMCHECK (http://valgrind.org/info/tools.html#memcheck), a well-known memory error detector for C and C++ programs based on Valgrind [21]. MEMCHECK can identify a broad range of memory errors: illegal memory accesses (*e.g.,* heap over- and under-runs, stack overflows, and accesses to freed memory), uses of undefined variables, erroneous heap-memory handling (*e.g.,* double frees, mismatches between memory allocations and deallocations), pointer overlaps in `memcpy` and other related functions, and memory leaks.

2) Objects of study: For our study, we used three real and widely-used C programs: `flex`, `grep`, and `sed`. We obtained these programs, together with test suites for each of them, from the SIR repository (http://sir.unl.edu/portal/). Table I provides, for each program, a concise description (Description), information about its size (KLOC), its number of test cases (# Tests), and the coverage achieved by the test cases (Coverage).

3) Experiment data: To collect the data needed for our study, we first ran MEMCHECK on each program P and computed $memerr(I)$, the set of memory errors detected by MEMCHECK when running P against its test suite I. This error set includes all the memory errors that can be detected on P given the tool and inputs considered. For `flex`, `grep`, and `sed`, the size of $memerr(I)$ is 2, 7, and 5, respectively.

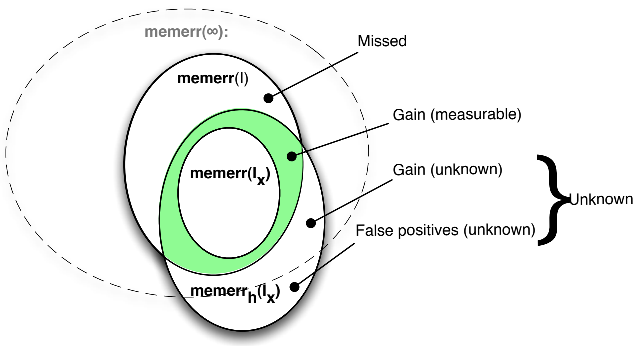


Figure 3. Intuitive view of our experimental settings for the study.

Next, for each program P, we created different subsets of its test suite I as follows. We generated all possible subsets S_K of P's $memerr(I)$ set and, for each S_K, we identified the set $I_{S_k} \subseteq I$ of test inputs that reveal all and only the memory errors in S_K. We then discarded the I_{S_k} subsets that included all test inputs in I, which resulted in a final number of 2, 24, and 8 I_{S_k} sets for `flex`, `grep`, and `sed`.

Then, for each subject program P and each I_{s_k} for P, we computed $memerr(I_{s_k})$, the set of memory errors detected by MEMCHECK when executed on P for only the inputs in I_{s_k}. We also computed $memerr_h(I_{s_k})$, the set of memory errors revealed by execution hijacking when executed on the same set of inputs. We actually computed $memerr_h(I_{s_k})$ in five different ways, using execution hijacking (1) with no infeasibility-mitigation technique enabled, (2) with only filtering enabled, (3) with only early termination enabled, (4) with only state fixing enabled, and (5) with all infeasibility-mitigation techniques enabled.

Figure 3 shows an intuitive view of the data we just described, analogous to the view of execution hijacking provided in Figure 2. In the figure, $memerr(\infty)$ indicates the ideal set of all possible memory errors of P—the set of memory errors that would be exposed by running the program with all possible inputs in its domain. Set $memerr(I) \subset memerr(\infty)$ is the set of memory errors that we know because they are reported by MEMCHECK when it analyzes normal executions of P against its test suite I. Set $memerr(I_x) \subseteq memerr(I)$ is the set of memory errors revealed by normal executions of the subset of test inputs $I_x \subset I$. Finally, set $memerr_h(I_x)$ is the set of memory errors computed by the hijacked executions of I_x.

The errors in $(memerr(I) \cap memerr_h(I_x) - memerr(I_x))$, indicated as $Gain(measurable)$ in Figure 3, represent the gain in memory error detection provided by execution hijacking. Because such errors are in $memerr(I)$, they are definitely true positives, and because they are not in $memerr(I_x)$, they are not discovered without execution hijacking. Conversely, the memory errors in $(memerr(I) - memerr_h(I_x))$ are true errors that execution hijacking fails to identify; we call this set $Missed$.

The hijacked executions would typically also reveal additional memory errors that do not appear in $memerr(I)$ and can be computed as the set $(memerr_h(I_x) - memerr(I))$. Such errors can include both additional true positives, indicated as $Gain(unknown)$ in the figure, and false positives, indicated as $False\ positives(unknown)$. Distinguishing between these two cases requires a manual examination of the results, which can be a difficult and error-prone task for somebody who is not familiar with the program at hand. Therefore, we conservatively consider $Gain(unknown)$ and $False\ positives(unknown)$ together as a set that we call $Unknown$, which represents an upper bound on the number of false positives that execution hijacking can generate when applied to memory error detection.

B. Results

The data described in the previous section allows us to answer the two research questions that we are investigating in our study. We discuss each research question separately.

1) RQ1: RQ1 investigates whether execution hijacking can increase the effectiveness of dynamic-analysis techniques. To address this question, we use the data described in Section IV-A3. More precisely, for each program P considered, we compute the average of $Gain(measurable)$ and $Missed$ across all of the subsets $I_x \subset I$ for P.

In this case, we consider the data collected for execution hijacking run without any infeasibility-mitigation technique enabled, as we are interested in assessing the maximum gain that can be provided by the approach. (We present the results of RQ2, which investigates the trade-offs between gain and infeasibility, in Section IV-B2.)

Figure 4 presents the values of $memerr(I_x)$, $Gain(measurable)$, and $Missed$ for the three programs. As the figure shows, execution hijacking always increases the number of (real) memory errors detected by MEMCHECK. For `sed`, MEMCHECK without execution hijacking detects 48% of the known memory errors, whereas it detects 93% of such errors when run with execution hijacking. For `grep`, the percentage of detected memory errors increases from 51% to 57%, which is the lowest gain among the three subjects. Finally, for `flex`, the percentage of detected errors increases from 50% to 75%.

To get a better understanding of the results, we also checked the percentage of the individual subsets of test inputs I_x for which execution hijacking provided a benefit. We found that execution hijacking was able to increase the number of memory errors revealed by a subset of inputs in 50%, 33%, and 100% of the cases for `flex`, `grep`, and `sed`, respectively.

Overall, these results are encouraging and illustrate the potential of execution hijacking to improve dynamic analysis results. Although NIONKA is more effective on some programs than others, it increased the number of memory errors detected by MEMCHECK for all of the programs considered. Across programs, NIONKA was able to improve memory error detection for 17 of the 34 input subsets considered, and in most of these cases, the improvements were considerable.

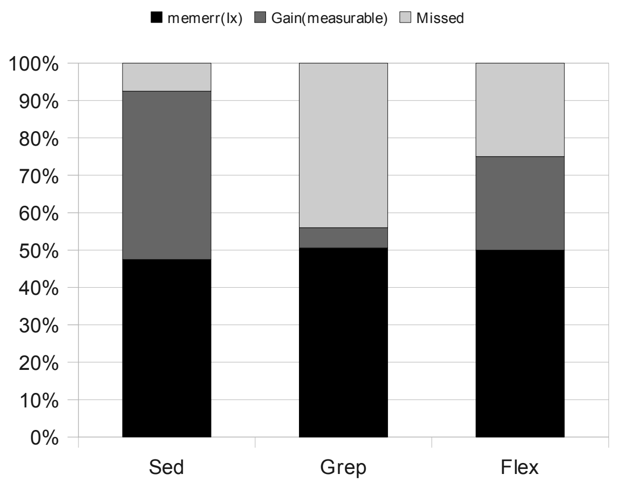


Figure 4. Effectiveness of execution hijacking: additional errors detected and missed errors.

2) RQ2: The results for RQ1 show the effectiveness of execution hijacking but do not mention the unknowns (*i.e.,* the upper bound for the number of false positives) that a dynamic analysis based on execution hijacking can produce when observing infeasible behaviors. In RQ2, we consider such unknowns and investigate whether our infeasibility-mitigation techniques (see Section III-B) can reduce the number of infeasible behaviors generated by execution hijacking, while still providing benefits. Intuitively, when applying those techniques, there exists a trade-off between the gain achieved by execution hijacking and the number of unknown errors reported—although the techniques can reduce the number of unknowns, they can also reduce the gain.

Figure 5 presents, as a segmented bar chart, the data about the memory errors identified using NIONKA in the five configurations discussed in Section IV-A3: (1) without any infeasibility-mitigation technique enabled (original), (2) with only filtering enabled (filter), (3) with only early termination enabled (termination), (4) with only state fixing enabled (fixing), and (5) with all infeasibility-mitigation techniques enabled (combined).

For each subject, the figure contains five bars, one for each configuration, and each bar contains four segments. The bottom segment (Removed TPs) for a given configuration represents the percentage of true positives generated by NIONKA that are eliminated when the infeasibility-mitigation techniques for that configuration are applied. This percentage is zero in most cases, so the first segment is present only in a few bars. The second segment from the bottom (TPs) is the percentage of true positives revealed by NIONKA that are not affected by the mitigation techniques for that configuration. The third segment (Unknowns) is the percentage of unknowns that the mitigation techniques are not able to eliminate. Finally, the top segment (Removed unknowns)

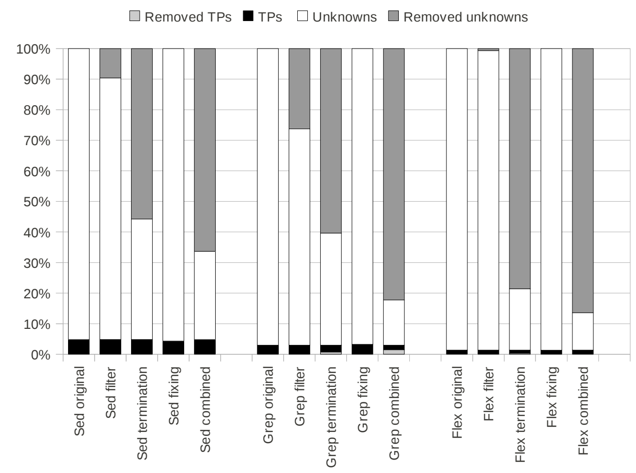


Figure 5. Impact of infeasibility-mitigation techniques on the results of execution hijacking.

represents the percentage of unknowns that the mitigation techniques do remove. Therefore, the top two segments, considered together, represent the percentage of unknown memory errors generated by MEMCHECK when run with NIONKA and without any mitigation technique enabled.

As the data shows, the total number of unknowns produced when using execution hijacking without mitigation (original) is fairly large compared to the gain provided, which will likely affect the practical usefulness of the technique. In other words, the gains are not large enough to compensate the effort required to separate true positives and false positives manually. In such cases, developers would not be able to take advantage of execution hijacking to improve dynamic analysis due to excessive noise in the results.

The data also shows that our infeasibility-mitigation techniques can reduce the percentage of unknowns without affecting the gains in most cases. Among the individual techniques, early termination is the most successful at reducing unknowns. This result confirms the intuition that preventing inconsistent states from propagating too extensively can considerably reduce the amount of infeasible behaviors generated.

Conversely, state fixing is the least-effective technique, with basically no effect on the number of unknowns. This may be an artifact of the fact that we performed a limited form of state fixing that targets scalar variables only (see Section III-B3). We therefore plan to investigate, in future work, more powerful approaches for state fixing that can handle conditions involving pointers and non-primitive type values.

Filtering's effectiveness is also fairly limited. It provides some help in reducing the number of unknowns for `sed` and, especially, `grep`, but it has almost no effect on `flex`.

Finally, the combined use of all three infeasibility-mitigation techniques provides the best results overall, which is not surprising considering that the three techniques are mostly orthogonal.

Overall, we consider also the results for RQ2 to be encouraging. Although the total number of unknowns is still larger than the number of true positives, which is not ideal, such number is dramatically reduced by the mitigation techniques. Most importantly, the gains provided by execution hijacking are almost unaffected by the mitigation techniques. Based on these findings, and considering that this was our first attempt at improving execution hijacking, we believe that it is possible to define more aggressive mitigation techniques that can further improve the ratio between true positive and unknowns. Furthermore, the number of unknowns is an *upper bound* on the number of false positives, and in fact we expect at least some of the unknowns to be additional true positives.

C. Threats to Validity

The most significant threats to the validity of our results are threats to external validity, which arise when the observed results cannot be generalized to other experimental setups. In our study of memory errors, we used three C programs. Therefore, we can draw only limited conclusions on how our results might generalize to other programs. Also, our evaluation considered only one type of dynamic analysis—memory error detection. The performance of execution hijacking may vary for other kinds of analysis. However, the programs we used are real, widely used programs, and the analysis considered is a very popular one, which mitigates such threats of external validity.

Threats to internal validity arise when factors affect the dependent variables (the data described in Section IV-A3) without the researchers' knowledge. In our study, such factors are errors in the implementation of the technique. One of the strengths of execution hijacking, especially in its current formulation, is that it does not require sophisticated analysis to be performed, which reduces the risk for implementation errors. Moreover, we extensively tested NIONKA on small examples to gain confidence in its implementation.

V. Related Work

The conventional way of exploring new program behaviors is based on test-input generation. Although many advances have been made in automated test-input generation, especially based on techniques that use symbolic execution (*e.g.,* [12], [14], [15]) or a combination of symbolic and concrete execution (*e.g.,* [16], [17]), test-input generation continues to be an expensive and inherently extremely complex task. Systematic exploration of a large number of behaviors using test-input generation is therefore not practical.

Researchers have also investigated ways to observe additional program behaviors by monitoring and/or sampling field executions (*e.g.,* [1], [9]–[11]). The benefit of such techniques is that the field executions represent how the software is actually used, instead of how it was anticipated to be used. Thus, the representativeness of developer-provided test inputs can be significantly enriched. However, practical considerations still limit the applicability of such techniques. Among these, performance and privacy concerns restrict the amount of instrumentation that can be added to, and the data that can be collected from, deployed software.

Execution hijacking is an alternative technique for observing new program behaviors without monitoring field executions or generating new test inputs. The idea of forcibly altering the outcome of a predicate at runtime has been investigated in the specific contexts of identifying elided conditionals [23], fault localization [24], malware analysis [18], [19], and fault detection [20]. This previous work, however, targeted specific contexts and, in some cases, did not share our goal of exploring new program behaviors.

Renieris and colleagues [23] introduced the notion of an elided conditional: a conditional statement whose evaluation is inconsequential. To identify such conditionals, their technique forcibly changes the value of a condition and monitors for differences in the program output. Their goal is not to explore new program behaviors, but rather to identify predicates that have no effect on the program outcome.

Zhang and colleagues [24] present a fault-localization technique that is based on switching a predicate at runtime. Given a failing execution that produces an incorrect output, their technique re-executes the program and switches the outcome of an executed predicate in each execution, until either the correct output is produced or all predicates have been switched. Also in this case, the goal of predicate switching is not to explore unseen behaviors, but to identify a program path that can generate a correct output.

Moser and colleagues [18] present, in the context of malware analysis, a technique that uses a single execution to explore multiple program paths. Their goal is to detect malicious program behavior that is difficult to identify using test-input generation (*e.g.,* a malware could perform a malicious activity only on certain dates). Specifically, their approach drives program execution based on how certain inputs (*e.g.,* the content of a file or the current system time) are used to decide the outcome of a predicate p. By forcing alternative evaluations of p, their approach can explore whether the program could exhibit malicious behaviors had those external (environmental) conditions been satisfied. Their approach performs consistent state updates, to the extent possible, at p to ensure that the path traversed after flipping p does not start from an inconsistent state.

Related to the previous approach is the approach by Wilhelm and Chiueh [19]. They present a system, LIMBO, that uses forced sampled execution to detect kernel rootkits. In a single execution of a kernel driver, LIMBO flips predicates based on a depth-first traversal of the driver's control-flow graph. Unlike the approach of Moser and colleagues, LIMBO does not perform state updates to ensure consistency.

Lu and colleagues [20] present a hardware-supported approach, called PATHEXPANDER, that executes "not-taken" paths to increase path coverage, with the goal of improving fault detection. PATHEXPANDER attempts to reduce the state inconsistency along a not-taken path by fixing the variables used in the condition at which a non-taken path begins. Although related to our approach, the main emphasis of their work is on leveraging hardware support to enable the use of their technique on user platforms.

Finally, techniques that dynamically modify program executions have also been explored for identifying regression faults [25], supporting fault localization [26], detecting concurrency-related faults [27], and assess software robustness through fault injection [28].

The key distinguishing aspect of our work from all these approaches is that we investigate execution hijacking as a general technique for observing new program behaviors—without generating new program inputs—that can be applied to different types of dynamic analyses. Specifically, we have evaluated the effectiveness of hijacking for memory-error detection and, in our extended technical report [29], for software testing. Moreover, unlike most existing approaches, we propose several techniques for mitigating the infeasibility problem inherent in execution hijacking.

VI. CONCLUSION AND FUTURE WORK

In this paper we presented an approach, called execution hijacking, for improving the effectiveness of dynamic analysis. Execution hijacking forces execution to proceed along unexplored paths by flipping the outcome of predicates at runtime. Although a hijacked execution is infeasible, it can reveal behaviors that could occur in a normal execution with a different input. In such cases, hijacking can enable dynamic-analysis techniques to observe new behaviors without requiring the generation of new program inputs. Execution hijacking can therefore overcome the limitations imposed by the cost and complexity of input generation, which restricts dynamic analyses to observe only a small fraction of a program's behavior.

We also presented a prototype tool, NIONKA, that implements execution hijacking for C and C++ programs and that is freely available.[2] Using NIONKA, we performed an empirical evaluation in which we used NIONKA to improve the effectiveness of memory error detection, a commonly performed dynamic analysis. Our results, albeit still preliminary, are promising: for the programs and inputs considered, execution hijacking was able to improve the results of the dynamic analysis. Our results also indicate that hijacking can reveal many unknown behaviors that may not be easy to classify as true or false positives. However, we provided initial evidence that suitably defined infeasibility-mitigation techniques can considerably reduce the number of unknown behaviors.

Although encouraging, our results are just a first step in this challenging research area, and many interesting directions are still to be explored. In our investigation, we performed a limited evaluation of three different techniques for reducing possibly infeasible behaviors introduced by execution hijacking. Further evaluation of these approaches and investigation of improved techniques—such as more sophisticated state-fixing techniques or techniques that leverage program dependences to perform more aggressive filtering—are needed to provide stronger evidence of the usefulness of execution hijacking. Also, our current instantiation of execution hijacking makes certain choices about how many branches to flip in an execution, which branches to flip, and when to flip; other choices for these aspects need to be investigated. In the immediate future, we will perform additional empirical evaluation of the approach by considering more subjects and, especially, more dynamic analyses. In particular, we are currently considering the use of execution hijacking to support dynamic invariant detection [30] and specification mining [7].

Overall, we believe that, if our initial results were to be confirmed by further experimentation, and improved with the development of more effective infeasibility-mitigation techniques, execution hijacking could represent a considerable step forward in the area of dynamic analysis.

Acknowledgments

This work was supported in part by NSF awards CCF-0916605 and CCF-0725202 to Georgia Tech.

References

[1] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable Statistical Bug Isolation,” in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 15–26.

[2] M. Jump and K. McKinley, “Cork: Dynamic memory leak detection for garbage-collected languages,” in *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007, pp. 31–38.

[3] N. Mitchell and G. Sevitsky, “Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications,” in *Proc. of the 17th European Conference on Object-Oriented Programming*, 2003, pp. 351–377.

[4] W.-K. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang, “Profile-guided proactive garbage collection for locality optimization,” in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 332–340.

[5] A. Orso, T. Apiwattanapong, and M. J. Harrold, “Leveraging field data for impact analysis and regression testing,” in *Proc. of the 9th European Software Engineering Conference and 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, september 2003, pp. 128–137.

[6] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, “Quickly detecting relevant program invariants,” in *Proc. of the 22nd International Conference on Software Engineering*, 2000, pp. 449–458.

[7] G. Ammons, R. Bodík, and J. R. Larus, “Mining specifications,” in *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 4–16.

[8] J. Bowring, A. Orso, and M. J. Harrold, “Monitoring deployed software using software tomography,” in *Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, November 2002, pp. 2–9.

[9] J. Clause and A. Orso, “A Technique for Enabling and Supporting Debugging of Field Failures,” in *Proc. of the 29th IEEE and ACM SIGSOFT International Conference on Software Engineering*, 2007, pp. 261–270.

[10] The GAMMA Project, http://gamma.cc.gatech.edu/, georgia Institute of Technology.

[11] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, “Applying classification techniques to remotely-collected program execution data,” in *Proc. of the 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 146–155.

[12] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[13] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[14] W. Visser, C. S. Păsăreanu, and S. Khurshid, “Test Input Generation with Java PathFinder,” *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[15] T. Xie, D. Marinov, W. Schulte, and D. Notkin, “Symstra: A framework for generating object-oriented unit tests using symbolic execution,” in *Proc. of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 365–381.

[16] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[17] K. Sen, D. Marinov, and G. Agha, “CUTE: A Concolic Unit Testing Engine for C,” in *Proc. of the 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.

[18] A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” in *Proc. of the IEEE Symposium on Security and Privacy*, 2007, pp. 231–245.

[19] J. Wilhelm and T. Chiueh, “A forced fampled execution approach to kernel rootkit identification,” in *Proc. of the 10th International Symposium on Recent Advances in Intrusion Detection*, ser. LNCS, vol. 4637, 2007, pp. 219–235.

[20] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas, “PathExpander: Architectural support for increasing the path coverage of dynamic bug detection,” in *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006, pp. 38–52.

[21] J. Seward and N. Nethercote, “Using Valgrind to detect undefined value errors with bit-precision,” in *Proc. of the 2005 USENIX Annual Technical Conference*, 2005.

[22] S. McConnell, “Daily build and smoke test,” *IEEE Software*, vol. 13, no. 4, p. 144, 1996.

[23] M. Renieris, S. Chan-Tin, and S. P. Reiss, “Elided conditionals,” in *Proc. of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Jun. 2004, pp. 52–57.

[24] X. Zhang, N. Gupta, and R. Gupta, “Locating faults through automated predicate switching,” in *Proc. of the 28th International Conference on Software Engineering*, May 2006, pp. 272–281.

[25] J. Laski, W. Szermer, and P. Luczycki, “Dynamic mutation testing,” in *Proc. of the 15th International Conference on Software Engineering*, May 1993, pp. 108–117.

[26] A. Zeller, “Isolating cause-effect chains from computer programs,” in *Proc. of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 2002, pp. 1–10.

[27] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, “Multithreaded Java program test generation,” *IBM Systems Journal*, vol. 41, no. 1, pp. 111–125, 2002.

[28] J. Voas and G. McGraw, *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley & Sons, 1997.

[29] P. Tsankov, W. Jin, A. Orso, and S. Sinha, “Execution hijacking: Improving dynamic analysis by flying off course,” Georgia Institute of Technology, Tech. Rep. GIT-CERCS-10-12, 2010.

[30] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 1–25, Feb. 2001.