Active Property Checking

Patrice Godefroid
Microsoft Research
Redmond, WA 98052
USA
pg@microsoft.com

Michael Y. Levin
Microsoft Center for Software Excellence
Redmond, WA 98052
USA
mlevin@microsoft.com

David Molnar
UC Berkeley and Microsoft Research
Berkeley, CA 94720
USA
dmolnar@cs.berkeley.edu

ABSTRACT

Runtime property checking (as implemented in tools like Purify or Valgrind) checks whether a program execution satisfies a property. *Active property checking* extends runtime checking by checking whether the property is satisfied by *all* program executions that follow the same program path. This check is performed on a symbolic execution of the given program path using a constraint solver. If the check fails, the constraint solver generates an alternative program input triggering a new program execution that follows the same program path but exhibits a property violation. Combined with systematic dynamic test generation, which attempts to exercise all feasible paths in a program, active property checking defines a new form of dynamic software model checking (program verification). In this paper, we formalize and study active property checking. We show how static and dynamic type checking can be extended with active type checking. Then, we discuss how to implement active property checking efficiently. Finally, we discuss results of experiments with media playing applications on Windows, where active property checking was able to detect several new security-related bugs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, Algorithms, Reliability, Security

1. INTRODUCTION

Today's embedded systems are exposed to "untrusted" inputs as never before, raising serious security concerns. Mobile phones have web browsers and connect to the Internet, where they encounter adversarial web pages which attempt to exploit web browser flaws to take control of visitors' devices. Personal media players, car stereos, and home stereo systems play media files downloaded by the user from arbitrary sites on the Internet, or, in some cases, directly connect to remote servers to enjoy streaming media. Even more restricted embedded systems, such as legacy SCADA systems developed in the context of closed, closely monitored networks, may be placed on open wireless connections or even connected to the Internet. In all these cases, software bugs which formerly were only stability issues, such as buffer overflow errors in parsing, become methods by which an adversary can take control of the embedded system. For example, one method used to remove the iPhone's protections against third-party software was the exploitation of an integer overflow vulnerability in TIFF image parsing [1]. In this paper, we discuss new test generation techniques to find such security-critical bugs.

Code inspection is one of the primary ways serious security bugs can be found and fixed before deployment, but manual code inspection is extremely expensive. During the last decade, code inspection for standard programming errors has largely been automated with static code analysis. Commercial static program analysis tools (e.g., [4, 14]) are now routinely used in many software development organizations. These tools are popular because they find many real software bugs, thanks to three main ingredients: they are *automatic*, they are *scalable*, and they check *many properties*. Intuitively, any tool that is able to check automatically (with good enough precision) millions of lines of code against hundreds of coding rules and properties is bound to find on average, say, one bug every thousand lines of code.

Our research goal is to automate, as much as possible, an even more expensive part of the software development process, namely *software testing*, which usually accounts for about 50% of the R&D budget of software development organizations. In particular, we want to automate *test generation* by leveraging recent advances in program analysis, automated constraint solving, and the increasing computation power available on modern computers. Compared to static analysis, test generation has a key benefit: the test case itself demonstrates the presence of a bug; there are no "false positives." To replicate the success of static program analysis in the testing space, we need the same key ingredients: automation, scalability and the ability to check many properties.

Automating test generation from program analysis is an old idea [18, 21], and work in this area can roughly be partitioned into two groups: static versus dynamic test genen-

EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.

eration. *Static test generation* [18, 2, 25, 7] consists of analyzing a program statically to attempt to compute input values to drive its executions along specific program paths. In contrast, *dynamic test generation* [19, 9, 5, 11] consists in executing the program, typically starting with some random inputs, while simultaneously performing a symbolic execution to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer program executions along alternative program paths. Since dynamic test generation extends static test generation with additional runtime information, it can be more powerful [9]. Scalability in this context has been recently addressed in [8, 11]. The motivation of the present work is to address the third challenge, which has been largely unexplored so far: how to dynamically check many properties simultaneously, thoroughly and efficiently, in order to maximize the chances of finding bugs during an automated testing session?

Traditional runtime checking tools like Purify [16], Valgrind [23] and AppVerifier [6] check a *single* program execution against a set of properties (such as the absence of buffer overflows, uninitialized variables or memory leaks). We show in this paper how such traditional *passive* runtime property checkers can be extended to *actively* search for property violations. Consider the simple program:

```
int divide(int n,int d) { // n and d are inputs
  return (n/d); // division-by-zero error if d==0
}
```

The program `divide` takes two integers `n` and `d` as inputs and computes their division. If the denominator `d` is zero, an error occurs. To catch this error, a traditional runtime checker for division-by-zero would simply check whether the concrete value of `d` satisfies `(d==0)` just before the division is performed in that specific execution, but would not provide any insight or guarantee concerning other executions. Testing this program with random values for `n` and `d` is unlikely to detect the error, as `d` has only one chance out of 2^{32} to be zero if `d` is a 32-bit integer. Static and dynamic test generation techniques that attempt to cover specific or all feasible paths in a program will also likely miss the error since this program has a single program path which is covered no matter what inputs are used. However, the latter techniques could be helpful provided that a test `if (d==0) error()` was inserted before the division `(n/d)`: they could then attempt to generate an input value for `d` that satisfies the constraint `(d==0)`, now present in the program path, and detect the error. This is essentially what *active property checking* does: it injects *at runtime* additional symbolic constraints that, when solvable by a constraint solver, will generate new test inputs leading to property violations.

In other words, *active property checking* extends runtime checking by checking whether the property is satisfied by *all* program executions that follow the same program path. This check is performed on a dynamic symbolic execution of the given program path using a constraint solver. If the check fails, the constraint solver generates an alternative program input triggering a new program execution that follows the same program path but exhibits a property violation. We call this "active" checking because a constraint solver is used to "actively" look for inputs that cause a runtime check to fail. Combined with systematic dynamic test generation, which attempts to exercise all feasible paths in a program, active property checking defines a new form of program verification for terminating programs.

Closely Related Work. Checking properties at runtime on a dynamic symbolic execution of the program was suggested in [20], but may return false alarms whenever symbolic execution is imprecise, which is often the case in practice. Active property checking extends the idea of [20] by combining it with constraint solving and test generation in order to further check using a new test input whether the property is actually violated as predicted by the prior imperfect symbolic execution. This way, no false alarms are ever reported.

Although ideas similar to active property checking have been independently mentioned briefly in two recent papers [5, 17], prior work on dynamic test generation provides no study of active property checking, no formalization, no connections with static and dynamic type checking, no optimizations, and no evaluation with a representative set of checkers. [5] reports several bugs, but does not specify which were found using active property checking (which is only alluded to in one paragraph on page 3). While [17] discusses adding assertions to check properties in the program, it focuses on "predicting" property violating executions, not on test case generation, and their experiments report only "predicted" errors, as done in [20], which we specifically aim to extend.

Contributions. Our work aims to provide the first comprehensive study of this simple, general, yet largely unexplored idea of active property checking. Specifically, our work makes several contributions:

- We formalize active property checking semantically in Section 3 and show how it can provide a form of program verification when combined with (sound and complete) systematic dynamic test generation (recalled in Section 2). The technical report associated with this work in addition shows how active type checking connects with traditional static and dynamic type checking [10].

- Section 4 discusses how to implement active checking efficiently by minimizing the number of calls to the constraint solver, minimizing formula sizes and using two constraint caching schemes.

- Section 5 describes our implementation of 13 active checkers for security testing of media playing Windows applications in conjunction with systematic dynamic test generation of x86 binaries. We stress that while our implementation targets x86, the main ideas apply to any instruction set and architecture. Results of experiments searching for bugs in these media playing applications are discussed in Section 6. Active property checking was able to detect several new bugs in those applications.

Note that dynamic test generation is complementary to static analysis because it can analyze code that current static analysis does not handle well (e.g., due to function pointers or inline assembly). The new bugs found in codec applications studied in our paper had been missed by static analysis. Active property checking also requires no programmer intervention or annotations.

2. SYSTEMATIC DYNAMIC TEST GENERATION

Dynamic test generation (see [9] for further details) consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables x and expressed in terms of input parameters α. Side-by-side concrete and symbolic executions are performed using a concrete store Δ and a symbolic store Σ, which are mappings from program variables to concrete and symbolic values respectively. A *symbolic value* is any expression sv in some theory $\mathcal{T}$ where all free variables are exclusively input parameters α. For any variable x, $\Delta(x)$ denotes the *concrete value* of x in Δ, while $\Sigma(x)$ denotes the symbolic value of x in Σ. The judgment $\Delta \vdash e \rightarrow v$ means an expression e reduces to a concrete value v, and similarly $\Sigma \vdash e \rightarrow sv$ means that e reduces to a symbolic value sv. For notational convenience, we assume that $\Sigma(x)$ is always defined and is simply $\Delta(x)$ by default if no expression in terms of inputs is associated with x. The notation $\Delta(x \mapsto c)$ denotes updating the mapping Δ so that x maps to c.

The program P manipulates the memory (concrete and symbolic stores) through statements, or *commands*, that are abstractions of the machine instructions actually executed. A command can be an *assignment* of the form $x := e$ (where x is a program variable and e is an expression), a *conditional statement* of the form `if` e `then` C `else` C' where e denotes a boolean expression, and C and C' are *continuations* denoting the unique next statement to be evaluated (programs considered here are thus sequential and deterministic), or `stop` corresponding to a program error or normal termination.

Given an input vector $\vec{\alpha}$ assigning a value to every input parameter α, the evaluation of a program defines a unique finite *program execution* $s_0 \xrightarrow{C_1} s_1 \ldots \xrightarrow{C_n} s_n$ that executes the finite sequence $C_1 \ldots C_n$ of commands and goes through the finite sequence $s_1 \ldots s_n$ of program states. Each *program state* is a tuple $\langle C, \Delta, \Sigma, pc \rangle$ where C is the next command to be evaluated, and pc is a special meta-variable that represents the current path constraint. For a finite sequence w of statements (i.e., a control path w), a *path constraint* pc_w is a formula of theory $\mathcal{T}$ that characterizes the input assignments for which the program executes along w. To simplify the presentation, we assume that all the program variables have some default initial concrete value in the initial concrete store Δ_0, and that the initial symbolic store Σ_0 identifies the program variables v whose values are program inputs (for all those, we have $\Sigma_0(v) = \alpha$ where α is some input parameter). We also assume that all program executions eventually terminate. Initially, pc is defined to `true`.

Systematic dynamic test generation [9] consists of systematically exploring all feasible program paths of the program under test by using path constraints and a constraint solver. By construction, a path constraint represents conditions on inputs that need be satisfied for the current program path to be executed. Given a program state $\langle C, \Delta, \Sigma, pc \rangle$ and a constraint solver for theory $\mathcal{T}$, if C is a conditional statement of the form `if` e `then` C `else` C', any satisfying assignment to the formula $pc \wedge sv$ (respectively $pc \wedge \neg sv$) defines program inputs that will lead the program to execute the `then` (resp. `else`) branch of the conditional statement. By systematically repeating this process, such a *directed search* can enumerate all possible path constraints and eventually execute all feasible program paths.

```
1 int buggy(int x) { // x is an input
2   int buf[20];
3   buf[30]=0; // buffer overflow independent of x
4   if (x > 20)
5     return 0;
6   else
7     return buf[x]; // buffer overflow if x==20
8 }
```

Figure 1: Example of program with buffer overflows.

The search is exhaustive provided that the generation of the path constraint (including the underlying symbolic execution) and the constraint solver for the given theory $\mathcal{T}$ are both *sound and complete*, that is, for all program paths w, the constraint solver returns a satisfying assignment for the path constraint pc_w *if and only if* the path is feasible (i.e., there exists some input assignment leading to its execution). In this case, in addition to finding errors such as the reachability of bad program statements (like `assert(0)`), a directed search can also prove their absence, and therefore obtain a form of program *verification*.

THEOREM 1. *(adapted from [9]) Given a program P as defined above, a directed search using a path constraint generation and a constraint solver that are both sound and complete exercises all feasible program paths exactly once.*

In this case, if a program statement has not been executed when the search is over, this statement is not executable in any context.

In practice, path constraint generation and constraint solving are usually not sound and complete. When a program expression cannot be expressed in the given theory $\mathcal{T}$ decided by the constraint solver, it can be simplified using concrete values of sub-expressions, or replaced by the concrete value of the entire expression. For example, if the solver handles only linear arithmetic, symbolic sub-expressions involving multiplications can be replaced by their concrete values.

3. ACTIVE CHECKERS

Even when sound and complete, a directed search based on path exploration alone can miss errors that are not path invariants, i.e., that are not violated by *all* concrete executions executing the same program path, or errors that are not caught by the program's run-time environment. This is illustrated by the simple program shown in Figure 1.

This program takes as (untrusted) input an integer value stored in variable `x`. A buffer overflow in line 3 will be detected at run-time only if a runtime checker monitors buffer accesses. Such a runtime checker would thus check whether any array access of the form `a[x]` satisfies the condition $0 \leq \Delta(\mathtt{x}) < b$ where $\Delta(\mathtt{x})$ is the concrete value of array index `x` and b denotes the bound of the array `a` (b is 20 for the array `buf` in the example of Figure 1). Let us call such a traditional runtime checker for concrete values a *passive checker*.

Moreover, a buffer overflow is also possible in line 7 provided `x==20`, yet a directed search focused on path explo-

ration alone may miss this error. The reason is that the only condition that will appear in a path constraint for this program is `x > 20` and its negation. Since most input values for `x` that satisfy `¬(x > 20)` do not cause the buffer overflow, the error will likely be undetected with a directed search as defined in the previous section.

In order to catch the buffer overflow on line 7, the program should be extended with a *symbolic* test $0 \leq \Sigma(\mathtt{x}) < b$ (where $\Sigma(\mathtt{x})$ denotes the symbolic value of array index `x`) just before the buffer access `buf[x]` on line 7. This will force the condition $0 \leq \mathtt{x} < 20$ to appear in the path constraint of the program in order to refine the partitioning of its input values. An *active checker* for array bounds can be viewed as systematically adding such symbolic tests before all array accesses.

Formally, we define *passive checkers* and *active checkers* as follows.

DEFINITION 1. *A* passive checker *for a property π is a function that takes as input a finite program execution w, and returns $\mathtt{fail}_\pi$ iff the property π is violated by w.*

Because we assume all program executions terminate, properties considered here are *safety* properties. Runtime property checkers like Purify [16], Valgrind [23] and AppVerifier [6] are examples of tools implementing passive checkers.

DEFINITION 2. *Let pc_w denote the path constraint of a finite program execution w. An active checker for a property π is a function that takes as input a finite program execution w, and returns a formula ϕ_C such that the formula $pc_w \wedge \neg\phi_C$ is satisfiable iff there exists a finite program execution w' violating property π and such that $pc_{w'} = pc_w$.*

Active checkers can be implemented in various ways, for instance using property monitors/automata, program rewrite rules or type checking. They can use private memory to record past events (leading to the current program state), but they are not allowed any side effect on the program. Section 5 discusses detailed examples of *how* active checkers can be implemented. Here are examples of specifications of active checkers.

EXAMPLE 1. ***Division By Zero.*** *Given a program state where the next statement involves a division by a denominator `d` which depends on an input (i.e., such that $\Sigma(\mathtt{d}) \neq \Delta(\mathtt{d})$), an active checker for division by zero outputs the constraint $\phi_{Div} = (\Sigma(\mathtt{d}) \neq 0)$.*

EXAMPLE 2. ***Array Bounds.*** *Given a program state where the next statement involves an array access `a[x]` where `x` depends on an input (i.e., is such that $\Sigma(\mathtt{x}) \neq \Delta(\mathtt{x})$), an active checker for array bounds outputs the constraint $\phi_{Buf} = (0 \leq \Sigma(\mathtt{x}) < b)$ where b denotes the bound of the array `a`.*

EXAMPLE 3. ***NULL Pointer Dereference.*** *Consider a program expressed in a language where pointer dereferences are allowed. Given a program state where the next statement involves a pointer dereference `*p` where `p` depends on an input (i.e., such that $\Sigma(\mathtt{p}) \neq \Delta(\mathtt{p})$), an active checker for NULL pointer dereference generates the constraint $\phi_{NULL} = (\Sigma(\mathtt{p}) \neq \mathtt{NULL})$.*

Multiple active checkers can be used simultaneously by simply considering separately the constraints they inject in a given path constraint. Such a way, they are guaranteed not to interfere with each other (since they have no side effects). We will discuss how to combine active checkers to maximize performance in Section 4.

By applying an active checker for a property π to all feasible paths of a program P, we can obtain a form of *verification* for this property, that is stronger than Theorem 1.

THEOREM 2. *Given a program P as defined above, if a directed search (1) uses a path constraint generation and constraint solvers that are both sound and complete,* and *(2) uses both a passive and an active checker for a property π in all program paths visited during the search, then the search reports $\mathtt{fail}_\pi$ iff there exists a program input that leads to a finite execution violating ϕ.*

Proof Sketch: Assume there is an input assignment that leads to a finite execution w of P violating π. Let pc_w be the path constraint for the execution path w. Since path constraint generation and constraint solving are both sound and complete, we know by Theorem 1 that w will eventually be exercised with some concrete input assignment $\vec{\alpha}'$. If the passive checker for π returns $\mathtt{fail}_\pi$ for the execution of P obtained from input $\vec{\alpha}'$ (for instance, if $\vec{\alpha}' = \vec{\alpha}$), the proof is finished. Otherwise, the active checker for π will generate a formula ϕ_C and call the constraint solver with the query $pc_w \wedge \neg\phi_C$. The existence of $\vec{\alpha}'$ implies that this query is satisfiable, and the constraint solver will return a satisfying assignment from which a new input assignment $\vec{\alpha}''$ is generated ($\vec{\alpha}''$ could be $\vec{\alpha}$ itself). By construction, running the passive checker for π on the execution obtained from that new input $\vec{\alpha}''$ will return $\mathtt{fail}_\pi$. □

Note that both passive checking and active checking are required in order to obtain this result, as illustrated earlier with the example of Figure 1. In practice, however, symbolic execution, path constraint generation, constraint solving, passive and active property checking are typically not sound and complete, and therefore active property checking reduces to testing.

4. OPTIMIZATIONS

4.1 Minimizing Calls to the Constraint Solver

As discussed in Section 3, (negations of) constraints injected by various active checkers in a same path constraint can be solved independently one-by-one since they have no side effects. We call this a *naive combination* of checker constraints.

However, the number of calls to the constraint solver can be reduced by bundling together constraints injected at the same or equivalent program states into a single conjunction. If pc denotes the path constraint for a given program state, and $\phi_{C1}, \ldots, \phi_{Cn}$ are a set of constraints injected in that state by each of the active checkers, we can define the combination of these active checkers by injecting the formula $\phi_C = \phi_{C1} \wedge \cdots \wedge \phi_{Cn}$ in the path constraint, which will result in the single query $pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$ to the constraint solver. We can also bundle in the same conjunction constraints ϕ_{Ci} injected by active checkers at different program states anywhere in between two conditional statements, i.e., anywhere between two constraints in the path constraint (since those program states are indistinguishable by that path constraint). This combination reduces the number of calls to the constraint solver but, if the query

Procedure `CombineActiveCheckers`$(I, pc, \phi_{C1}, \ldots, \phi_{Cn})$:
1. Let $x =$ `Solve`$(pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn}))$
2. If $x =$ `UNSAT` return I
3. For all i in $[1, n]$, eliminate ϕ_{Ci} if x satisfies $\neg\phi_{Ci}$
4. Let $\phi_{C1}, \ldots, \phi_{Cm}$ denote the remaining ϕ_{Ci} $(m < n)$
5. If $m = 0$, return $I \cup \{x\}$
6. Call `CombineActiveCheckers`$(I \cup \{x\}, pc, \phi_{C1}, \ldots, \phi_{Cm})$

Figure 2: Function to compute a strongly-sound combination of active checkers.

$pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$ is satisfied, a satisfying assignment produced by the constraint solver may not satisfy *all* the disjuncts, i.e., may violate only *some* of the properties being checked. Hence, we call this a *weakly-sound combination*.

A *strongly-sound*, or *sound* for short, combination can be obtained by making additional calls to the constraint solver using the simple function shown in Figure 2. By calling

`CombineActiveCheckers`$(\emptyset, pc, \phi_{C1}, \ldots, \phi_{Cn})$

the function returns a set I of input values that covers *all* the disjuncts that are satisfiable in the formula $pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$. The function first queries the solver with the disjunction of all the checker constraints (line 1). If the solver returns `UNSAT`, we know that all these constraints are unsatisfiable (line 2). Otherwise, we check the solution x returned by the constraint solver against each checker constraint to determine which are satisfied by solution x (line 3). (This is a model-checking check, not a satisfiability check; in practice, this can be implemented by calling the constraint solver with the formula $\bigwedge_i (b_i \Leftrightarrow \neg\phi_{Ci}) \wedge pc \wedge (\bigvee_i b_i)$ where b_i is a fresh boolean variable which evaluates to `true` iff $\neg\phi_{Ci}$ is satisfied by a satisfying assignment x returned by the constraint solver; determining which checker constraints are satisfied by x can then be done by looking up the values of the corresponding bits b_i in solution x.) Then, we remove these checker constraints from the disjunction (line 4), and query the solver again until all checker constraints that can be satisfied have been satisfied by some input value in I. If t out of the n checkers can be satisfied in conjunction with the path constraint pc, this function requires at most $\texttt{min}(t+1, n)$ calls to the constraint solver, because each call removes at least one checker from consideration. Obtaining strong soundness with fewer than t calls to the constraint solver is not possible in the worse case. Note that the naive combination defined above is strongly-sound, but always requires n calls to the constraint solver.

It is worth emphasizing that none of these combination strategies attempt to minimize the number of input values (solutions) needed to cover all the satisfiable disjuncts. This could be done by querying first the constraint solver with the *conjunction* of all checker constraints to check whether any solution satisfies all these constraints simultaneously, i.e., to check whether their intersection is non-empty. Otherwise, one could then iteratively query the solver with smaller and smaller conjunctions to force the solver to return a minimum set of satisfying assignments that cover all the checker constraints. Unfortunately, this procedure may require in the worse case $O(2^n)$ calls to the constraint solver. (The problem can be shown to be NP-complete by a reduction from the NP-hard SET-COVER problem.)

Weakly and strongly sound combinations capture possible overlaps, inconsistencies or redundancies between active checkers at equivalent program states, but is independent of how each checker is specified: it can be applied to any active checker that injects a formula at a given program state. Also, the above definition is independent of the specific reasoning capability of the constraint solver. In particular, the constraint solver may or may not be able to reason precisely about combined theories (abstract domains and decision procedures) obtained by combining individual constraints injected by different active checkers. Any level of precision is acceptable with our framework and is an orthogonal issue (e.g., see [12]). Whether the constraint solver supports incremental solving is also orthogonal.

4.2 Minimizing Formulas

Minimizing the number of calls to the constraint solver should not be done at the expense of using longer formulas. Fortunately, the above strategies for combining constraints injected by active checkers also reduce formula sizes.

For instance, consider a path constraint pc and a set of n constraints $\phi_{C1} \ldots \phi_{Cn}$ to be injected at the end of pc. The naive combination makes n calls to the constraint solver, each with a formula of length $|pc| + |\phi_{Ci}|$, for all $1 \leq i \leq n$. In contrast, the weak combination makes only a single call to the constraint solver with a formula of size $|pc| + \Sigma_{1 \leq i \leq n} |\phi_{Ci}|$, i.e., a formula (typically much) smaller than the sum of the formula sizes with the naive combination. The strong combination makes, in the worse case, n calls to the constraint solver with formulas of size $|pc| + \Sigma_{1 \leq i \leq j} |\phi_{Ci}|$ for all $1 \leq j \leq n$, i.e., possibly bigger formulas than the naive combination. But often, the strong combination makes fewer calls than the naive combination, and matches the weak combination in the best case (when none of the disjuncts $\neg\phi_{Ci}$ are satisfiable).

In practice, path constraints pc tend to be long, much longer than injected constraints ϕ_{Ci}. A simple optimization (implemented in [9, 5, 24, 11]) consists of eliminating the constraints in pc which do not share symbolic variables (including by transitivity) with the negated constraint c to be satisfied. This *unrelated constraint elimination* can be done syntactically by constructing an undirected graph G with one node per constraint in $pc \cup \{c\}$ and one node per symbolic (input) variable such that there is an edge between a constraint and a variable iff the variable appears in the constraint. Then, starting from the node corresponding to constraint c, one performs a (linear-time) traversal of the graph to determine with constraints c' in pc are reachable from c in G. At the end of the traversal, only the constraints c' that have been visited are kept in the conjunction sent to the constraint solver, while the others are eliminated.

With unrelated constraint elimination and the naive checker combination, the size of the reduced path constraint pc_i may vary when computed starting from each of the n constraints ϕ_{Ci} injected by the active checkers. In this case, n calls to the constraint solver are made with the formulas $pc_i \wedge \neg\phi_{Ci}$, for all $1 \leq i \leq n$. In contrast, the weak combination makes a single call to the constraint solver with the formula $pc' \wedge (\vee_i \neg\phi_{Ci})$ where pc' denotes the reduced path constraint computed when starting with the constraint $\vee_i \neg\phi_{Ci}$. It is easy to see that $|pc'| \leq \Sigma_i |pc_i|$, and therefore that the formula used with the weak combination is again smaller than the sum of the formula sizes used with the naive

```
1 #define k 100 // constant
2 void Q(int *x, int a[k]){ // inputs
3   int tmp1,tmp2,i;
4   if (x == NULL) return;
5   for (i=0; i<=k;i++) {
6     if (a[i]>0) tmp1 = tmp2+*x;
7     else tmp2 = tmp1+*x;
8   }
9   return;
10}
```

Figure 3: A program with $O(2^k)$ possible execution paths. A naive application of a NULL dereference active checker results in $O(k \cdot 2^k)$ additional calls to the constraint solver, while local constraint caching eliminates the need for any additional calls to the constraint solver.

combination. Loosely speaking, the strong combination includes again both the naive and weak combinations as two possible extremes.

4.3 Caching Strategies

No matter what strategy is used for combining checkers at a single program point, constraint *caching* can significantly reduce the overhead of using active checkers.

To illustrate the benefits of constraint caching, consider a NULL dereference active checker (see Section 3) and the program Q in Figure 3. Program Q has $2^k + 1$ executions, where 2^k of those dereference the input pointer `x` k times each. A naive approach to dynamic test generation with a NULL dereference active checker would inject k constraints of the form `x` $\neq$ `NULL` at each dereference of `*x` during every such execution of Q, which would result in a total of $k \cdot 2^k$ additional calls to the constraint solver (i.e., k calls for each of those executions).

To limit this expensive number of calls to the constraint solver, a first optimization consists of *locally caching* constraints in the current path constraint in such a way that syntactically identical constraints are never injected more than once in any path constraint. (Remember path constraints are simply conjunctions.) This optimization is applicable to any path constraint, with or without active checkers. The correctness of this optimization is based on the following observation: if a constraint c is added to a path constraint pc, then for any longer pc' extending pc, we have $pc' \Rightarrow pc$ (where $\Rightarrow$ denotes logical implication) and $pc' \wedge \neg c$ will always be unsatisfiable because c is in pc'. In other words, adding the same constraint multiple times in a path constraint is pointless since only the negation of its first occurrence has a chance to be satisfiable.

Constraints generated by active checkers can be dealt with by injecting those in the path constraint like regular constraints. Indeed, for any constraint c injected by an active checker either at the end of a path constraint pc or at the end of a longer path constraint pc' (i.e., such that $pc' \Rightarrow pc$), we have the following:

- if $pc \wedge \neg c$ is unsatisfiable, then $pc' \wedge \neg c$ is unsatisfiable;
- conversely, if $pc' \wedge \neg c$ is satisfiable, then $pc \wedge \neg c$ is satisfiable (and has the same solution).

Therefore, we can check $\neg c$ as early as possible, i.e., in conjunction with the shorter pc, by inserting the first occurrence of c in the path constraint. If an active checker injects the same constraint later in the path constraint, local caching will simply remove this second redundant occurrence.

By injecting constraints generated by active checkers into regular path constraints and by using local caching, a given constraint c, like `x` $\neq$ `NULL` in our previous example, will appear at most once in each path constraint, and a single call to the constraint solver will be made to check its satisfiability for each path, instead of k calls as with the naive approach without local caching. Moreover, because the constraint `x` $\neq$ `NULL` already appears in the path constraint due to the `if` statement on line 4 before any pointer dereference `*x` on lines 6 or 7, it will never be added again to the path constraint with local caching, and no additional calls will be made to the constraint solver due to the NULL pointer dereference active checker for this example.

Another optimization consists of caching constraints *globally* [5]: whenever the constraint solver is called with a query, this query and its result are kept in a (hash) table shared between execution paths during a directed search. In Section 6, the effect of both local and global caching is measured empirically.

5. IMPLEMENTATION

We implemented active checkers as part of a dynamic test generation tool called SAGE (*Scalable, Automated, Guided Execution*) [11]. SAGE uses the `iDNA` tool [3] to trace executions of Windows programs, then virtually re-executes these traces with the `TruScan` trace replay framework [22]. During re-execution, SAGE checks for file read operations and marks the resulting bytes as symbolic. As re-execution progresses, SAGE generates symbolic constraints for the path constraint. After re-execution completes, SAGE uses the constraint solver `Disolver` [15] to generate new input values that will drive the program down new paths. SAGE then completes this cycle by testing and tracing the program on the newly generated inputs. The new execution traces obtained from those new inputs are sorted by the number of new code blocks they discover, and the highest ranked trace is expanded next to generate new test inputs and repeat the cycle [11]. While we describe in the technical report version of this paper how to combine static analysis with active property checking, the current SAGE implementation does not perform any static analysis [10].[1]

The experiments reported in the next section were performed with 13 active checkers, shown in Figure 4 with an identifying number. The number 0 refers to a constraint generated by observing a branch on tainted data, as in basic DART or EXE, that becomes part of the path constraint. Number 1 refers to a division-by-zero checker, 2 denotes a NULL pointer dereference checker, and 4 and 5 denote array underflow and overflow checkers (see Section 3). Number 3 refers to an active checker that looks for function arguments

[1]Static analysis of the codec applications discussed next is problematic due to function pointers and in-line assembly code.

Number	Checker	Number	Checker
0	Path Exploration	7	Integer Underflow
1	DivByZero	8	Integer Overflow
2	NULL Deref	9	MOVSX Underflow
3	SAL NotNull	10	MOVSX Overflow
4	Array Underflow	11	Stack Smash
5	Array Overflow	12	AllocArg Underflow
6	REP Range	13	AllocArg Overflow

Figure 4: Active checkers implemented.

that have been annotated with the `notnull` attribute in the SAL property language [13], and attempts to force those to be NULL. Checker type 6 looks for the x86 `REP MOVS` instruction, which copies a range of bytes to a different range of bytes, and attempts to force a condition where the ranges overlap, causing unpredictable behavior. Checkers 7 and 8 are for integer underflows and overflows. Checkers type 9 and 10 target the `MOVSX` instruction, which sign-extends its argument and may lead to loading a very large value if the argument is negative. The "stack smash" checker, type 11, attempts to solve for an input that directly overwrites the stack return pointer, given a pointer dereference that depends on a symbolic input. Finally, checkers type 12 and 13 look for heap allocation functions with symbolic arguments; if found, they attempt to cause overflow or underflow of these arguments.

An active checker in SAGE first registers a `TruScan` callback for specific events that occur during re-execution. For example, an active checker can register a callback that fires each time a symbolic input is used as an address for a memory operation. The callback then inspects the concrete and symbolic state of the re-execution and decides whether or not to emit an active checker constraint. If the callback does emit such a constraint, SAGE stores it in the current path constraint.

SAGE implements a *generational search* [11]: given a path constraint, all the constraints in that path are systematically negated one-by-one, placed in a conjunction with the prefix of the path constraint leading it, and attempted to be solved with the constraint solver. Constraints injected by active checkers are inserted in the path constraint and treated as regular constraints during a generational search.

Because we work with x86 machine-code traces, some information we would like to use as part of our active checkers is not immediately available. For example, when SAGE observes a `load` instruction with a symbolic offset during re-execution, it is not clear what the bound should be for the offset. We work around these limitations by leveraging the `TruScan` re-execution infrastructure. During re-execution, `TruScan` observes calls to known allocator functions. By parsing the arguments to these calls and their return values, as well as detecting the current stack frame, `TruScan` builds a map from each concrete memory address to the bounds of the containing memory object. We use the bounds associated with the memory object pointed to by the concrete value of the address as the upper and lower bound for an active bounds check of the memory access. Therefore, source code or debug symbols are not required, although they may be used if available.

Media 1	none	weak	strong	naive
Total Time (s)	16	37	42	37
Solver Time (s)	5	5	10	5
# Tests Gen	59	70	87	105
# Disjunctions	N/A	11	11	N/A
Dis. Min/Mean/Max	N/A	2/4.2/16	2/4.2/16	N/A
# Path Constr.	67	67	67	67
# Checker Constr.	N/A	46	46	46
# Solver Calls	67	78	96	113
Max CtrList Size	77	141	141	141
Mean CtrList Size	2.7	2.7	2.7	3
Local Cache Hit	79%	81%	88%	88%
Media 2	none	weak	strong	naive
Total Time (s)	761	973	1140	1226
Solver Time (s)	421	463	601	504
# Tests Gen	1117	1833	2734	5122
# Disjunctions	N/A	1125	1125	N/A
Dis. Min/Mean/Max	N/A	1/5.4/216	1/5.4/216	N/A
# Path Constr.	3001	2990	2990	2990
# Checker Constr.	N/A	6080	6080	6080
# Solver Calls	3001	4115	5368	9070
Max CtrList Size	11141	91739	91739	91739
Mean CtrList Size	368	373	373	372
Local Cache Hit	39%	19.5%	19.5%	19.5%

Figure 6: Microbenchmark statistics.

6. EVALUATION

We report results of experiments with active checkers and two applications which play media files and are widely used on Windows. Figure 5 shows the result of a single symbolic execution and test generation task for each of these two test programs. The second column indicates which checkers injected constraints during that program execution. The last column gives the number of symbolic input bytes read during that single execution, which is 100 to 1,000 times larger than previously reported with dynamic test generation [9, 5, 24].

For each application, we ran microbenchmarks to quantify the marginal cost of active checking during a single symbolic execution task and measure the effectiveness of our optimizations. We then performed long-running searches with active checkers to investigate their effectiveness at finding bugs. These searches were performed on a 32-bit Windows Vista machine with two dual-core AMD Opteron 270 processors running at 2 GHz, with 4 GB of RAM and a 230 GB hard drive; all four cores were used in each search. We now describe observations from these experiments. We stress that these observations are from a limited sample size and should be taken with caution.

6.1 Microbenchmarks

Figure 6 presents statistics for our two test programs obtained with a single symbolic execution and test generation task with no active checkers, or the weak, strong and naive combinations of active checkers discussed in Section 4. For each run, we report the total run time (in seconds), the time spent in the constraint solver (in seconds), the number of test generated, the number of disjunctions bundling together checker constraints (if applicable) before calling the constraint solver, the minimum, mean, and maximum number

Test	Checkers Injected	Time (secs)	*pc* size	# checker constraints	# Tests	# Instr.	Symbolic Input Size
Media 1	0,2,4,5,7,8	37	67	46	105	3795771	65536
Media 2	0,1,2,4,5,7,8,9,10,11	1226	2990	6080	5122	279478553	27335

Figure 5: Statistics from a *single* symbolic execution and test generation task with a naive combination of all 13 checkers. We report the checker types that injected constraints, the total time for symbolic execution test generation, the number of constraints in the total path constraint, the total number of injected checker constraints, the number of tests generated, the number of instructions executed after the first file read, and the number of symbolic input bytes.

of constraints in disjunctions (if applicable[2]). We also report the total number of constraints in the path constraint, the total number of constraints injected by checkers, the number of calls made to the constraint solver, statistics about the size needed to represent all path and checker constraints (discussed further below) and the local cache hit. Each call to the constraint solver was set with a timeout value of 5 seconds, which we picked because almost all queries we observed terminated within this time.

Checkers produce more test cases than path exploration at a reasonable cost. As expected, using checkers increases total run time but also generates more tests. For example, all checkers with naive combination for Media 2 creates 5122 test cases in 1226 seconds, compared to 1117 test cases in 761 seconds for the case of no active checkers; this gives us 4.5 times as many test cases for 61% more time spent in this case. As expected (see Section 4), the naive combination generates more tests than the strong combination, which itself generates more tests than the weak combination. Perhaps surprisingly, most of the extra time is spent in symbolic execution, not in solving constraints. This may explain why the differences in runtime between the naive, strong and weak cases are relatively not that significant. Out of curiosity, we also ran experiments (not shown here) with a "basic" set of checkers that consisted only of Array Bounds and DivByZero active checkers; this produced fewer test cases, but had little to no runtime penalty for test generation for both test programs.

Weak combination has the lowest overhead. We observed that the solver time for weak combination of disjunctions was the lowest for Media 2 runs with active checkers and tied for lowest with the naive combination for Media 1. The strong disjunction generates more test cases, but surprisingly takes longer than the naive combination in both cases. For Media 1, this is due to the strong combination hitting one more 5-second timeout constraints than the naive combination. For Media 2, we believe this is due to the overhead involved in constructing repeated disjunction queries. Because disjunctions in both cases have fairly few disjuncts on average (4 or 5), this overhead dominates for the strong combination, while the weak one is still able to make progress by handling the entire disjunction in one query.

For the SAGE system, the solver used requires less than a second for most queries. Therefore the time improvement for weak combination over naive combination in this microbenchmark is small, saving roughly 3 minutes, even though the number of calls to the solver is in fact reduced by almost 50%. Other implementations may obtain better time performance, especially if they more precisely model memory or include more difficult constraints. Both figures include the effect of other optimizations, such as caching, which we show are important for active checking.

Unrelated constraint elimination is important for checkers. Our implementation of the unrelated constraint optimization described in Section 5.2 introduces additional *common subexpression variables*. Each of these variable defines a subexpression that appears in more than one constraint. In the worst case, the maximum possible size of a list of constraints passed to our constraint solver is the sum of the number of these variables, plus the size of the path constraint, plus the number of checker constraints injected. We collected the maximum possible constraint list size (Max CtrList Size) and the mean size of constraint lists produced after our unrelated constraint optimization (Mean CtrList Size). The maximum possible size does not depend on our choice of weak, strong, or naive combination, but the mean list size is slightly affected. We observe in the Media 2 microbenchmarks that the maximum possible size jumps dramatically with the addition of checkers, but that the mean size stays almost the same. Furthermore, even in the case without checkers, the mean list size is 100 times smaller than the maximum. The Media 1 case was less dramatic, but still showed post-optimization constraint lists an order of magnitude smaller than the maximum. This shows that unrelated constraint optimization is key to efficiently implement active checkers.

6.2 Macrobenchmarks

For macrobenchmarks, we ran a generational search for 10 hours starting from the same initial media file, and generated test cases with no checkers, and with the weak and strong combination of all 13 checkers. We then tested each test case by running the program with AppVerifier [6], configured to check for heap errors. For each crashing test case, we recorded the checker kinds responsible for the constraints that generated the test. Since a search can generate many different test cases that exhibit the same bug, we "bucket" crashing files by the *stack hash* of the crash, which includes the address of the faulting instruction. We also report a bucket kind, which is either a NULL pointer dereference, a read access violation (ReadAV), or a write access violation (WriteAV). It is possible for the same bug to be reachable by program paths with different stack hashes for the same root cause. Our experiments always report the distinct stack hashes. We also computed the hit rate for global caching during each search.

Checkers can find bugs missed by path exploration. Figure 7 shows the crash buckets found for Media 2 by 10-hours searches with "No" active checkers, with a "W"eak and

[2]In the strong case, the mean number does not include disjunctions iteratively produced by the algorithm of Figure 2, which explains why the mean is the same as in the weak case.

Crash Bucket	Kind	0	2	4	5	7	8
1867196225	NULL	No/W/S				W/S	W/S
1867196225	ReadAV	No/W				W	W
1277839407	ReadAV	S					
1061959981	ReadAV		S			S	S
1392730167	ReadAV	S					
1212954973	ReadAV				S	S	S
1246509355	ReadAV				S	W/S	W/S
1527393075	ReadAV	S					
1011628381	ReadAV					S	W/S
2031962117	ReadAV	No/W/S					
286861377	ReadAV	No/S					
842674295	WriteAV		S	S		S	S

Figure 7: Crash buckets found for Media 1 by 10-hour searches with "No" active checkers, with a "W"eak and "S"trong combinations of active checkers. A total of 41658 **tests were generated and tested in** 30 **hours, with** 783 **crashing files in** 12 **buckets.**

Crash Bucket	Kind	0
790577684	ReadAV	No/W/S
825233195	ReadAV	No/W/S
795945252	ReadAV	No/W/S
1060863579	ReadAV	No/W

Figure 8: Crash buckets found for Media 2 by 10-hours searches with "No" active checkers, with a "W"eak and "S"trong combinations of active checkers. A total of 11849 **tests were generated and tested in** 30 **hours, with** 25 **crashing files in** 4 **buckets.**

"S"trong combinations of active checkers. For instance, an "S" in a column means that at least one crash in the bucket was found by the search with strong combination. The type of checkers whose constraint found the crash bucket is also indicated in the figure. For Media 1, the Null Deref (type 2) active checker found 2 crash buckets, the Array Underflow and Overflow (types 4 and 5) active checkers found 3 crash buckets, while the Integer Underflow and Overflow (types 7 and 8) active checkers found 7 crash buckets. Without any active checkers, the tool is able to find only 4 crash buckets in 10 hours of search, and misses the serious WriteAV bug detected by the strong combination only. For Media 2, in contrast, the test cases generated by active checkers did not find any new crash buckets, as shown in Figure 8.

Checker yield can vary widely. Figure 9 reports the overall number of injected constraints of each type during all 10-hours searches, and how many of those were successfully solved to create new test cases. It also reports the checker *yield*, or percentage of test cases that led to crashes. For Media 1, active checkers have a higher yield than test cases generated by path exploration (type 0). For Media 2, several checkers did inject constraints that were solvable, but their yield is 0% as they did not find any new bugs. The yield indicates how precise symbolic execution is. For Media 1, symbolic execution is very precise as every checker constraint violation for checker types 2, 4 and 5 actually leads to a crash (as is the case with a fully sound and complete constraint generation and solving as shown in Section 3); even if symbolic execution is perfect, the yield for the integer under/overflow active checkers may be less than 100% because not every integer under/overflow leads to a crash. In contrast, symbolic execution in our current implementation does not seem precise enough for Media 2, as yields are poor for this benchmark.

Media 1	0	2	4	5	7	8
Injected	27612	26	13	13	11153	11153
Solved	18056	22	2	3	3179	5552
Crashes	339	22	2	3	139	136
Yield	1.9%	100%	100%	100%	4.4%	2.4%

Media 2	0	1	2	4	5
Injected	13425	12	2146	1544	1551
Solved	4735	0	61	10	20
Crashes	7	0	0	0	0
Yield	1.4%	N/A	0%	0%	0%
	7	8	9	10	11
Injected	11158	11177	5	5	10
Solved	576	2355	0	5	0
Crashes	0	0	0	0	0
Yield	0%	0%	N/A	0%	N/A

Figure 9: Constraints injected by checker types, solved, crashes, and yield for Media 1 and Media 2, over both weak and strong combination 10-hours searches.

Local and global caching are effective. Local caching can remove a significant number of constraints during symbolic execution. For Media 1, we observed a 80% or more local cache hit rate (see Figure 6). For Media 2, the hit rates were less impressive but still removed roughly 20% of the constraints.

Our current implementation does not have a global cache for query results (see Section 4). To measure the impact of global caching on our macrobenchmark runs, we added code that dumps to disk the SHA-1 hash of each query to the constraint solver, and then computes the global cache hit rate. For Media 1, all searches showed roughly a 93% hit rate, while for Media 2 we observed 27%. This shows that there are significant redundancies in queries made by different test generation tasks during the same search.

7. CONCLUSIONS

The more one checks for property violations, the more one should find software errors. In this paper, we have defined and studied active property checking, a new form of dynamic property checking based on dynamic symbolic execution, constraint solving and test generation. We showed how active type checking extends traditional static and dynamic type checking. We presented several optimizations to implement active property checkers efficiently, and discussed results of experiments with several large shipped Windows applications. Active property checking was able to detect several new bugs in those applications.

Overall, we showed that without careful optimizations, active property checking can significantly slow down dynamic test generation. For the SAGE implementation, we found caching and unrelated constraint optimization to be the most important. While weak and strong combination did in fact reduce the number of calls to the constraint solver, this was not the bottleneck. Our results for Media 1 demonstrate that active property checking can find a significant number of bugs not discovered by path exploration alone.

At the same time, the results for Media 2 show that the current SAGE constraints do not fully capture the execution of some programs. Increasing constraint precision should lead to more expensive solver calls, therefore making our weak and strong combinations more important.

We have also performed exploratory searches on several other applications, including two shipped as part of Office 2007 and two media parsing layers. In one of the Office applications and media layer, the division by zero checker and the integer overflow checker each created test cases leading to previously-unknown division by zero errors. In the other cases, we also discovered new bugs in test cases created by checkers, but needed to use an internal tool for runtime passive checking of memory safety violations that is more precise than AppVerifier.

8. REFERENCES

[1] iphone TIFF image processing vulnerability, 2007. `http://secunia.com/advisories/27213/`.

[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Tests from Counterexamples. In *Proceedings of ICSE'2004 (26th International Conference on Software Engineering)*. ACM, May 2004.

[3] S. Bhansali, W. Chen, S. De Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments VEE*, 2006.

[4] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[6] Microsoft Corporation. AppVerifier, 2007. `http://www.microsoft.com/technet/prodtechnol/windows/appcompatibility/appverifier.mspx`.

[7] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *Proceedings of ICSE'2005 (27th International Conference on Software Engineering)*. ACM, May 2005.

[8] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages)*, pages 47–54, Nice, January 2007.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.

[10] P. Godefroid, M. Y. Levin, and D. Molnar. Active Property Checking. Technical report, Microsoft, 2007. MSR-TR-2007-91.

[11] P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, San Diego, February 2008. `http://research.microsoft.com/users/pg/public_psfiles/ndss2008.pdf`.

[12] S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proceedings of PLDI'2006 (ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation)*, Ottawa, June 2006.

[13] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.

[14] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *Proceedings of PLDI'2002 (2002 ACM SIGPLAN Conference on Programming Language Design and Implementation)*, pages 69–82, 2002.

[15] Y. Hamadi. Disolver: the distributed constraint solver version 2.44, 2006. `http://research.microsoft.com/~youssefh/DisolverWeb/disolver.pdf`.

[16] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 125–138, Berkeley, January 1992.

[17] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: Amplifying the effectiveness of software testing. Technical report, UC-Berkeley, April 2007. UCB/EECS-2007-35.

[18] J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

[19] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

[20] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of 12th USENIX Security Symposium*, Washington D.C., August 2003.

[21] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[22] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Languages Design and Implementation (PLDI)*, 2007.

[23] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[24] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering)*, Lisbon, September 2005.

[25] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04 (International Symposium on Software Testing and Analysis)*, Boston, July 2004.