Precise Pointer Reasoning for Dynamic Test Generation

Bassem Elkarablieh*
UT Austin
elkarabl@ece.utexas.edu

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Michael Y. Levin
Microsoft CSE
mlevin@microsoft.com

ABSTRACT

Dynamic test generation consists of executing a program while gathering symbolic constraints on inputs from predicates encountered in branch statements, and of using a constraint solver to infer new program inputs from previous constraints in order to steer next executions towards new program paths. Variants of this technique have recently been adopted in several bug detection tools, including our whitebox fuzzer SAGE, which has found dozens of new expensive security-related bugs in many Windows applications and is now routinely used in various Microsoft groups.

In this paper, we discuss how to perform precise symbolic pointer reasoning in the context of dynamic test generation. We present a new memory model for representing arbitrary symbolic pointer dereferences to memory regions accessible by a program during its execution, and show that this memory model is the most precise one can hope for in our context, under some realistic assumptions. We also describe how the symbolic constraints generated by our model can be solved using modern SMT solvers, which provide powerful constructs for reasoning about bit-vectors and arrays. This new memory model has been implemented in SAGE, and we present results of experiments with several large Windows applications showing that an increase in precision can often be obtained at a reasonable cost. Better precision in symbolic pointer reasoning means more relevant constraints and fewer imprecise ones, hence better test coverage, more bugs found and fewer redundant test cases.

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

Keywords

Software Testing, Automatic Test Generation, Pointer Reasoning, Program Verification

1. INTRODUCTION

Systematic dynamic test generation [7] consists of executing a program while gathering symbolic constraints on inputs from predicates encountered in branch statements, and of using a constraint solver to infer new program inputs from previous constraints in order to steer next executions towards *some* new program paths. This technique is now the foundation of several bug detection tools (see [3, 13, 11] among others), including our whitebox fuzzer SAGE [9]. SAGE can handle large applications and is optimized for handling long symbolic executions at the x86 binary level. Over the last 18 months, it has detected dozens of new expensive security bugs in many Windows applications. SAGE is so effective in finding bugs missed by other techniques like static analysis or blackbox random fuzzing that it is now used daily in various Microsoft groups.

Dynamic test generation tools vary by the type of programs they can analyze, the type of constraints their symbolic execution can generate and by the constraint solver they use. Whenever symbolic execution does not know how to generate a symbolic constraint for a program statement depending on some program inputs, the concrete values of those inputs can be used to simplify the constraint as a fallback [7]. However, these concretizations can result in failing to exercise program branches and paths, hence in missing bugs, and also in increasing the number of redundant test cases which *diverge* from their expected program path.

To illustrate this, consider generating test inputs for the method `single_array` in Figure 1 starting with the input tuple `{x = 0, y = 1}`. When the execution encounters the conditional statement `a[x] == a[y] + 2` involving the program inputs `x` and `y`, symbolically evaluating this constraint requires comparing the values at the symbolic memory addresses `&a[x]` and `&a[y]`. (Adopting C notation, `&v` denotes the memory location of program variable `v`.) If the symbolic execution is not able to reason about symbolic addresses, the concrete values `&a[0]` (since `0` is the concrete value of `x`) and `&a[1]` (since `1` is the concrete value of `y`) of those addresses can be used instead, and the constraint `a[x] == a[y]`

[*] The work of this author was done mostly while visiting Microsoft Research.

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

```
void single_array(BYTE x, BYTE y) {
    BYTE * a = new BYTE[4];

    a[0] = x;
    a[1] = 0;
    a[2] = 1;
    a[3] = 2;

    if (a[x] == a[y] + 2)
        assert(false);

    delete[] a;
}
```

Figure 1: An example of a symbolic memory dereference operation. Concretizing the value of a symbolic address results in adding imprecision to the symbolic analysis.

`+ 2` is then simplified into `(x != 2)`, which is stored in the path constraint symbolically representing the current execution. Solving the new path constraint `(x == 2)` obtained by negating this constraint results in a new input tuple `{x = 2, y = 1}` which is expected to exercise the other branch of the conditional statement and reach the `assert(false)` statement. However, executing `single_array` with inputs `{x = 2, y = 1}` actually results instead in taking the same program path as the previous one: we call this a *divergence*. Note that concretizing the symbolic addresses resulted in changing the semantics of the condition from trying to find two elements in the array that differ by 2, to assigning the value `2` to the input variable `x`; this imprecision resulted in both generating a redundant test input and missing a reachable statement.

In this paper, we discuss how to precisely reason about symbolic pointer dereferences in the context of dynamic test generation, in order to handle programs with constraints as in the previous example. A symbolic pointer dereference is a pointer dereference at an address whose value depends on the evaluation of a symbolic expression, i.e., depends on some (untrusted) inputs. For instance, `a[x]` is a symbolic pointer dereference at the symbolic address `&a[x]` (equivalently `&a + x`) depending on the value of input `x`. Unfortunately, a symbolic address `&a[x]` can potentially point to any memory location within its range of possible values (say 2^{32} if `x` is a 32-bit value), and considering all these possible values for each symbolic address dereference would be highly expensive and impractical. In contrast, the concrete address $M(e)$ of symbolic address e either points to a specific memory location that lies within a valid, well-defined region in memory, or points to an unallocated memory region, which is a *memory-access violation*.

Therefore, we propose a practical yet precise memory model based on the following insight. For each symbolic address e, we split the entire universe of possible addresses that e may take into two disjoint sets defined by the current concrete value $M(e)$ of that address: the set of concrete addresses contained in the valid memory region including the concrete address $M(e)$, and the rest of the universe. If the concrete address $M(e)$ does not point to a valid memory region, then a memory-access violation is reported. Otherwise, we consider two cases separately: we *assume* that the symbolic address e is confined within its valid memory region by forcing its value to be between the known concrete bounds defining the memory region; moreover, we also *assert* that the symbolic address e is confined within its valid memory region by checking separately whether its value could overflow or underflow the valid memory region, hence leading to a possible memory-access violation. This model can also be extended to deal with multiple pointer dereferences, where a single pointer may point to multiple valid memory regions.

The rest of the paper is organized as follows. In Section 2, we recall the basics of dynamic test generation and define under which assumptions it can be sound and complete. In Section 3, we present our new memory model and the steps necessary to implement it in conjunction with symbolic execution. In Section 4, we show that this memory model is *the most precise* one can hope for in our context, under some realistic assumptions. We then discuss related work in Section 5 and results of experiments in Section 6. In Section 7, we present an example of bug in a packet decoder which requires the precise symbolic pointer reasoning developed in this paper in order to be detected using dynamic test generation. We conclude in Section 8.

2. BACKGROUND: SYSTEMATIC DYNAMIC TEST GENERATION

Dynamic test generation (see [7] 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 M and a symbolic store S, which are mappings from *memory addresses* (where program variables are stored) to concrete and symbolic values respectively. A *symbolic value* is any expression e in some theory $\mathcal{T}$ where all free variables are exclusively input parameters α. For any program variable x, $M(x)$ denotes the *concrete value* of x in M, while $S(x)$ denotes the *symbolic value* of x in S. For notational convenience, we assume that $S(x)$ is always defined and is simply $M(x)$ by default if no expression in terms of inputs is associated with x in S. When $S(x)$ is different from $M(x)$, we say that that program variable x is "symbolic", meaning that the value of program variable x is a function of some input(s) which is represented by the symbolic expression $S(x)$ associated with x in the symbolic store. We also extend this notation to allow $M(e)$ to denote the concrete value of symbolic expression e when evaluated with the concrete store M. The notation $+$ for mappings denotes updating; for example, $M' = M + [m \mapsto v]$ is the same map as M, except that $M'(m) = v$.

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'' denote the unique next statement to be evaluated when e holds or does not hold, respectively (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 $\langle C, M, S, pc \rangle$ 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 unique initial concrete value in the initial concrete store M_0, and that the initial symbolic store S_0 identifies the program variables v whose values are program inputs (for all those, we have $S_0(v) = \alpha$ where α is the corresponding input parameter). We also assume that all program executions eventually terminate. Initially, pc is defined to `true`.

Systematic dynamic test generation [7] 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 $\langle C, M, S, pc \rangle$ 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 $pc \wedge e$ (respectively $pc \wedge \neg e$) 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.

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(false)`), a directed search can also prove their absence, and therefore obtain a form of program *verification*.

Theorem 1. *(adapted from [7]) 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.

Note that the above formalization and theorem do apply to programs containing loops or recursion, as long as all program executions terminate. However, in the presence of a single loop whose number of iterations depends on some unbounded input, the number of feasible program path becomes infinite, and exercizing all those paths (Theorem 1) will take forever. In practice, termination can always be forced by bounding input values, loop iterations or recursion, at the cost of potentially missing bugs.

```
evaluate_symbolic(e, M, S) =
  match e:
      case v: // Program variable v
        return S(&v)
      case &v: // Program variable v
        return evaluate_concrete(e, M)
      case +(e', e''): // Addition
        let f' = evaluate_symbolic(e',M,S)
        let f'' = evaluate_symbolic(e'',M,S)
        if f' and f'' are constants then
          return evaluate_concrete(e, M)
        else
          return create_expression('+',f',f'')
      case *e':
        let f' = evaluate_symbolic(e',M,S)
        if f' is a constant c then
          return S(*c)
        else
          // Old: return M(*evaluate_concrete(e', M))
          return get_dereference_expression(f',M,S)
      etc.
```

Figure 2: Symbolic expression evaluation.

3. A NEW MEMORY MODEL

We now show how path constraint generation can be made sound and complete in the presence of symbolic pointer dereferences. Throughout the rest of the paper, we use the term *symbolic address* e to refer to a symbolic expression in a given theory $\mathcal{T}$ that is used as an address, and *symbolic content* $*e$ to refer to the content stored at the symbolic address e. We also call $*e$ a *symbolic pointer dereference*, or *symbolic address dereference*. A symbolic address dereference that occurs during the symbolic execution of a conditional statement or of the right-hand side of an assignment statement is called a *symbolic read* operation, while a symbolic address dereference that occurs during the symbolic execution of the left-hand side of an assignment statement is called a *symbolic write* operation. The address where the content of a program variable v is stored is denoted by $\&v$.

Our approach for handling symbolic address dereferences is based on the following observations. A symbolic address e (say a 32-bit address) can potentially point to any memory location within its range of possible values. Considering *all* these 2^{32} possible values for each symbolic address dereference $*e$ would be highly expensive and impractical. In contrast, the concrete address $M(e)$ of symbolic address e either points to a specific memory location that lies within a valid, well-defined region in memory, or points to an unallocated memory region, which is a *memory-access violation*. A valid memory region is a region of memory with a known starting *address* and a *size*, which we assume here to be both input-independent, i.e., not symbolic (see Section 4). Such a region can reside in the heap, stack or data space of the running process.

Therefore, we propose a memory model which splits the entire universe of possible addresses that each symbolic address e may take into two disjoint sets defined by the concrete value $M(e)$ of that address: the set of concrete addresses contained in the valid memory region including the concrete address $M(e)$, and the rest of the universe. If the concrete address $M(e)$ does not point to a valid memory region, then a memory-access violation is reported. Otherwise, we consider two cases separately: we *assume* that the symbolic address e is confined within its valid memory region by forcing its value to be between $address$ and $address + size - 1$; moreover, we also *assert* that the symbolic address e is confined within its valid memory region by checking separately whether its value could overflow (be greater than $address + size - 1$) or underflow (be less than $address$), hence leading to a possible memory-access violation. Both cases can be implemented by simply injecting the new constraint $address \leq e < address + size$ in the current path constraint, in the style of "active property checking" described in [8].

This model can also be extended to deal with multiple pointer dereferences, such as $**e$. In this case, a *set* of valid memory regions can be associated with a single symbolic pointer dereference, as we will describe later.

To implement this memory model in conjunction with symbolic execution and path constraint generation, we need to (1) extend the symbolic execution to handle symbolic address dereferences, (2) determine the valid memory regions (if any) that may be pointed by a symbolic address, (3) generate constraints involving symbolic addresses and memory regions (including their contents), and (4) translate and then solve those constraints using a constraint solver. We next describe each of those steps in detail.

3.1 Symbolic Execution with Symbolic Address Dereferences

During the execution of program P, the unique next command C to be executed is determined by the current program state $\langle C, M, S, pc\rangle$ and is executed both concretely (as usual) and symbolically. Symbolic execution is performed by symbolically evaluating all expressions e involved in the command. For each such expression e, the function `evaluate_symbolic` shown in Figure 2 is invoked given the current concrete store M and the current symbolic store S. The function `evaluate_concrete` is used to compute the concrete value resulting from evaluating the expression with the concrete values of all its subexpressions. If an expression is a constant, then it has only a concrete value equal to that constant. If an expression is a program variable v, its symbolic value is the value stored at that address in the current symbolic store $S(\&v)$. If an expression is of the form $\&v$ for a program variable v, then the expression is evaluated concretely since program variables v are defined by the program and their addresses cannot be inputs. If an expression is a relational or arithmetic operation involving other subexpressions, like an addition, the subexpressions are recursively evaluated; if all subexpressions are evaluated to constants, the operation is executed with these concrete values and the result is returned; otherwise, a new symbolic expression representing the operation is returned.

If an expression is a memory dereference operation $*e'$, then the address e' is evaluated. If the address is a constant c, then its associated expression $S(*c)$ is returned from the symbolic memory store S. However, if the address is symbolic, a standard symbolic execution [7] would also concretize the address in this case and then return the concrete value stored at that concrete address (this is done in the commented line starting with "Old" in Figure 2). The latter approximation introduces imprecision in the symbolic execution.

This imprecision can be alleviated as follows. Whenever a symbolic address dereference $*e$ occurs during symbolic execution, the new function `get_dereference_expression` shown in Figure 3 is now called with the symbolic expression e. This function returns a symbolic expression representing all possible symbolic values that the symbolic pointer dereference $*e$ may return given the current concrete store M and symbolic store S.

To compute this expression, the function `get_dereference_expression` first calls the function `get_memory_regions` (see Figure 3) to determine the set of valid memory regions the symbolic address e may point to and obtain a memory *snapshot* for each of those regions. A snapshot for a valid memory region $\langle address, size\rangle$ is defined as the tuple $\langle address, size, m, s\rangle$ where m and s are snapshots of the concrete store M and symbolic store S within the addresses ranging from $address$ to $address{+}size{-}1$. Next, the function `add_bound_constraints` is called in order to add a set of constraints to bound the symbolic address within each of the valid memory regions. Finally, it creates an expression for the unary dereference operation `(*)` and associates it with the set of memory region snapshots the symbolic address may point to.

As an example, consider evaluating the expressions `a[x]` and `a[y]` in the function `single_array` of Figure 1. Figure 5 shows the expression tree constructed with our memory model while symbolically executing `single_array` with the input tuple `{x = 0, y = 1}`. The tree nodes labeled `N1` and `N2` represent the result of evaluating `a[x]` and `a[y]` respectively. Each node is now associated with a snapshot of memory corresponding to the possible results of the dereference. The snapshot records the start address, the size, as well as the memory content at that region. Snapshots are used later while updating the path constraint, adding bound constraints, and translating constraints to the SMT solver. In Figure 5, `X` represents an input parameter, while `x` represents a program variable. In what follows, we simplify the presentation and simply write `x` for both when there is a 1-to-1 mapping between a program input and the program variable where this input is being stored, which is always the case in the illustrative examples considered in this paper.

3.2 Identifying Valid Memory Regions

The function `get_memory_regions` shown in Figure 3 computes the set of possible valid memory regions a symbolic address e may point to by calling the function `expand_and_evaluate` of Figure 4, which either returns the concrete value of e in the case e is a pointer, or returns the set of concrete values e can take in the case e is a pointer dereference. The former case deals with symbolic pointer single-dereferences, which occurs when indexing uni-dimensional arrays, strings and object fields, while the latter case handles symbolic pointer multi-dereferences where a symbolic address depends on previous dereference operations, forming a chain of dereferences, as when indexing multi-dimensional arrays. For each address returned by `expand_and_evaluate`, the function `get_memory_region_snapshot` is called to compute a snapshot of the valid memory region including that address. We assume the function `get_region_info` keeps track of all valid memory regions currently allocated, either statically by the compiler, or dynamically by the memory manager during ex-

```
get_dereference_expression($e$) =
    let $region\_snapshots$ = get_memory_regions($e$)
    add_bound_constraints($e$, $region\_snapshots$)
    return create_expression('$*$', $e$, $region\_snapshots$)

get_memory_regions($e$) =
    let $region\_snapshots$ = $\emptyset$
    let $concrete\_values$ = expand_and_evaluate($e$)
    foreach $value \in concrete\_values$ do
        $regions\_snapshots$ = $region\_snapshots \cup$
                    get_memory_region_snapshot($value$)
    return $region\_snapshots$

get_memory_region_snapshot($address$) =
    let $< start, size >$ = get_region_info($address$)
    let $s$ = $\emptyset$
    let $m$ = $\emptyset$
    for $address$ : $start \rightarrow start + size - 1$ do
        $m = m + [address \mapsto M(address)]$
        $s = s + [address \mapsto S(address)]$
    return create_region_snapshot($start$, $size$, $m$, $s$)
```

Figure 3: Constructing a symbolic expression to represent a symbolic address content.

```
expand_and_evaluate($e$) =
    let result = $\emptyset$
    match $e$:
        case $v$: // Program variable $v$
            $result = M(\&v)$
        case $\&v$: // Program variable $v$
            $result = \&v$
        case $+(e', e'')$: // Addition
            let $f'$ = expand_and_evaluate($e'$)
            let $f''$ = expand_and_evaluate($e''$)
            foreach $x \in f'$ do
                foreach $y \in f''$ do
                    $result = result \cup (x + y)$
        case $*e'$:
            let $region\_snapshots$ = get_memory_regions($e'$)
            foreach $rs \in region\_snapshots$ do
                $result = result \cup$
                    get_region_concrete_values($rs$)
        etc.
    return $result$

get_region_concrete_values($rs$) =
    let $values$ = $\emptyset$
    for $address$:$rs.start \rightarrow rs.start + rs.size - 1$ do
        $values = values \cup rs.m(address)$
    return $values$
```

Figure 4: Partial evaluation of the possible concrete values for a symbolic address.

ecution. By construction, valid memory regions defined in our memory model never overlap and their content is always initialized with some default fixed value. If a given address is not included in any valid memory region, the function `get_region_info` detects it and reports a memory-access violation.

For instance, to identify the regions associated with `a[x]` in the `single_array` example, the `expand_and_evaluate` function is first called with `&a[x]` to determine a set of concrete addresses from different memory regions that `&a[x]` could take. Since `a[x]` is a uni-dimensional array access, the concrete value `&a[0]` of the address is returned. This value is then used by `get_memory_region_snapshot` to determine the region $\langle \&a, 4 \rangle$ as the only possible region that `&a[x]` can point to, and to capture the regions contents.

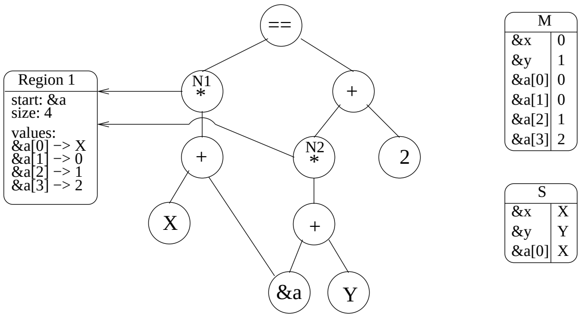

Figure 5: The symbolic expression tree for the `single_array` function of Figure 1. The unary (`*`) operator represent a symbolic memory dereference and is associated with a snapshot of the memory region pointed to at the time of the dereference.

3.3 Adding Bound Constraints

As previously discussed, our memory model both *assumes* and *asserts* that every symbolic address must lie within the bounds of a memory region. Both cases are implemented by adding a *bound constraint* in the current path constraint for every symbolic address dereference. These bound constraints are treated as regular constraints in a path constraint. During a directed search, those constraints are eventually negated and solved. A solution of a negated bound constraint represents a potential memory-access violation in the program under test.

Given a set of valid memory region snapshots $\{R_1, R_2, ..., R_n\}$ that may be pointed to by a symbolic address e, the function `add_bound_constraints` adds to the current path constraint a bound constraint of the form

$$\bigvee_i (e == e_i) \wedge (start_i \leq e < start_i + size_i)$$

where $start_i$ and $size_i$ denote the base address and size of region R_i, and where e_i denotes a symbolic expression such that $e == e_i$ holds if and only if e points to R_i. The symbolic expression e_i is computed by symbolically evaluating e except for the last level of dereference in the chain of dereferences defined by e. This computation is illustrated with the following examples.

3.4 Examples

Consider again the `single_array` example of Figure 1. Figure 5 shows the expression tree constructed with our memory model while executing the function `single_array` with the input tuple `{x = 0, y = 1}`. Instead of concretizing the dereference operations when evaluating the expression `a[x] == a[y] + 2`, each dereference operation (nodes *N1 and *N2) is now associated with a snapshot of memory corresponding to the possible results of the dereference. (Note that the expressions `*(&a[x])`, `*(&a + x)` and `a[x]` are all equivalent.) In order to cover the `then` branch of the conditional statement, the extended path constraint generated with our

```
void multi_array(BYTE x, BYTE y) {
    BYTE ** a = new BYTE*[2];
    a[0] = new BYTE[2];
    a[1] = new BYTE[3];

    a[0][0]=0; a[0][1]=1;
    a[1][0]=2; a[1][1]=3; a[1][2]=4;

    if (a[x][y] == y + 2)
        assert(false);

   delete[] a;
}
```

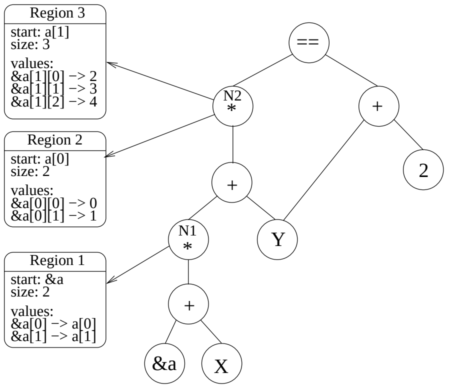

Figure 6: An example with a symbolic multi-dereference operation.

memory model is now the conjunction of the constraints:

```
c1:  a[0] == x
c2:  a[1] == 0
c3:  a[2] == 1
c4:  a[3] == 2
c5:  &a ≤ &a[x] < &a + 4
c6:  &a ≤ &a[y] < &a + 4
c7:  a[x] == a[y] + 2
```

These constraints are of three types: (1) constraints `{c1, c2, c3, c4}` represent the snapshot of the memory region $Region1$ associated with both dereferences `a[x]` and `a[y]` (Figure 5), (2) constraints `{c5, c6}` are the bound constraints for that region, and (3) constraint `{c7}` corresponds to the condition at the branch statement. A constraint solver can easily solve the conjunction of these constraints and generate the new test case `{x = 3, y = 1}` which results in covering the `assert(false)` statement. Additionally, by negating the constraints `{c5, c6}`, we can generate more test cases such as `{x = 10, y = 0}` and `{x = 0, y = 10}` that result in detecting buffer overflows in the program `single_array`.

Consider the second example `multi_array` shown in Figure 6. This program initializes a multi-dimensional array in the heap, and uses the input variables `x` and `y` to index the array. Figure 6 shows the expression tree constructed with our memory model while executing the function `multi_array` with the input tuple `{x = 0, y = 0}`. The expression `a[x][y]` is a chain of dereferences starting with `a[x]` and ending with `a[x][y]`. Thus `&a[x][y]` is equivalent to `a[x] + y`.

While symbolically executing `multi_array`, the symbolic dereference sub-expression `a[x]` is first encountered and evaluated using `get_dereference_expression`. Like in the case of `single_array`, `&a[x]` is associated with a single region $Region1\langle \&a, 2\rangle$, the bound constraint $\&a \leq \&a[x] < \&a + 2$ is added to the path constraint to force `&a[x]` within $Region1$, and a symbolic expression (node *N1) corresponding to the dereference operation is returned.

Next, the expression `a[x][y]` is evaluated with `get_dereference_expression`. First, `get_memory_regions` uses `expand_and_evaluate` to compute $\{$`a[0], a[1]`$\}$ as the set of possible concrete values for `&a[x]`, and uses these values in order to identify the two regions $Region2\langle a[0], 2\rangle$ and $Region3\langle a[1], 3\rangle$ as the possible valid memory regions that `&a[x][y]` may point to. Then, `add_bound_constraints` adds to the path constraint the following bound constraint:

$$\bigvee \begin{array}{l} (\texttt{\&a[x][y] == a[0] + y}) \wedge (\texttt{a[0]} \leq \texttt{\&a[x][y] < a[0] + 2}) \\ (\texttt{\&a[x][y] == a[1] + y}) \wedge (\texttt{a[1]} \leq \texttt{\&a[x][y] < a[1] + 3}) \end{array}$$

Finally, a symbolic expression (node *N2) corresponding to the multi-dereference operation is returned. The symbolic execution proceeds by building a symbolic expression for the condition of the branch statement, as previously described.

3.5 Handling Symbolic Write Operations

An assignment statement of the form $x := e$ consists of an expression e and of a destination address $\&x$ to store e. When executing an assignment statement, symbolic execution extended with our memory model first inspects the destination address. If the address is concrete, then the memory stores are updated, otherwise a symbolic write needs to be handled.

As in a symbolic read, a write operation using a symbolic address can potentially update any of the locations pointed to by the symbolic address. To handle a symbolic write, we first compute the regions that $\&x$ points to using the `get_dereference_expression` described earlier. Then, we associate a *symbolic update* $\&x \mapsto e$ with those regions.

After the regions are updated symbolically, values in those regions become nondeterministic, in the following sense: any subsequent read operation from any location in these regions is considered a symbolic read, even if the address used in the operation is concrete. For example, consider the code snippet below where `x` is an input variable:

```
BYTE a[4] = {1, 2, 3, 4};
a[x]=0;
if (a[3] == 0)
   assert(false);
```

While the expression `a[3]` does not involve any input-dependent expression, the result of `a[3]` is treated as a symbolic read as its outcome depends on the value of `x`: if `x` is 3 then 0 is returned, otherwise 4 is returned. To precisely model such cases, we treat `a[3]` as a symbolic read, and evaluates it using `get_dereference_expression` as described in Section 3.1.

3.6 Integration with the Z3 SMT Solver

The memory model presented in the previous sections has been implemented in SAGE [9]. To solve symbolic constraints, we use the Z3 [6] satisfiability modulo theory (SMT) solver. Z3 is a highly efficient SMT solver that targets program analysis problems. It supports several theories including integer arithmetic, fixed-size bit-vectors, arrays, and quantifiers. It also supports sign-extension operations which are very handy for modeling assembly language instructions.

```
z3_translate($e$, $memory$) =
    match $e$:
        case $c$: // a constant c
            return z3_bitvector(c)
        case $m$: // The symbolic variable m
            return z3_bitvector_var(m)
        case $+(e', e'')$: // Addition
            let $f'$ = z3_translate(e')
            let $f''$ = z3_translate(e'')
            return z3_bitvector_add(f', f'')
        case $*e'$: // a dereference expression
            // Translate the region snapshots associated with e'
            let $region\_snapshots = e'.region\_snapshots$
            foreach $region \in region\_snapshots$ do
                for $address$ : $region.start \rightarrow region.start + region.size - 1$ do
                    $memory$ = z3_store($memory$, $address$, $region.s(address)$)

            let $f'$ = z3_translate(e') // Translate the address
            return z3_select($memory$, $f'$)
        etc.
```

Figure 7: Translating a symbolic expression into a Z3 expression.

Array types in Z3 are maps from elements of a domain datatype to elements of a range datatype. Arrays are treated as un-interpreted functions, not as an ordered sequence of elements. Z3 provides two operations on maps, *select* and *store*. The *select* operation on a Z3 array takes a variable of the domain datatype and returns an existing element in array of the range type. The *store* operation on an array A takes an index i of the domain datatype and a value v of the range datatype and returns an array A' where $A' = A \text{ with } A[i] == v$.

A key advantage of using Z3 is its direct correlation with SAGE's memory model. We can rather easily translate SAGE's symbolic expressions, the memory regions associated with dereference expressions, and the bound constraints to Z3 primitives. Atomic expression representing the constants and the symbolic variable are modeled as 32 bit-vector Z3 terms. Memory is modeled as a Z3 array mapping 32 bit-vectors representing addresses into 8 bit-vectors representing the values.

The translation from SAGE to Z3 is illustrated by the `z3_translate` function shown in Figure 7. Primarily, atomic expression representing the constants and the symbolic variable are translated into 32 bit constants and variables. Binary arithmetic and relational operations are translated into the corresponding Z3 operations. Dereference operation require translating the updates (concrete or symbolic) that occurred on memory. For a dereference operation the address expression is first translated. Then for each region associated with the dereference expression, the updates performed on these regions are translated into *store* operations on memory. Finally the dereference operation is translated into a *select* operation from memory with the symbolic address as an index.

The result of the `z3_translate` on the symbolic expressions representing the program constraints is a Z3 boolean logic formula where, by construction [7], each free variable corresponds to a program input. If Z3 can solve the formula, it also generates a model satisfying the formula and this model can be transformed into a new input assignment to test the program further.

4. MAXIMUM PRECISION THEOREM

Consider a sequential deterministic program P as defined in Sections 2 and 3 and containing exclusively (assignment and conditional) statements $st(P)$ whose corresponding symbolic constraints are either expressions in a given decidable theory $\mathcal{T}$ (denoted $st(P) \subset \mathcal{T}$) for which there exists a sound and complete constraint solver, or are symbolic memory dereferences (read or write, single or multi dereferences).

Let such a program P be called *well-formed* if none of its executions can ever trigger a memory-access violation, i.e., a read or write operation outside all current valid memory regions, no matter where those regions reside. In other words, a well-formed program never triggers a memory-access violation even in the presence of an adversarial memory allocator.

Also, let us say that a valid memory region is *input-independent* if its starting *address* and *size* are both input-independent (i.e., do not depend on any input and therefore would only have a concrete value if tracked during symbolic execution).

The next theorem formally states that the memory model presented in this paper is *the most precise there is* for any program P as defined above whose valid memory regions are all input-independent.

THEOREM 2. *For any well-formed program P as defined above whose valid memory regions are all input-independent, the memory model and associated path constraint generation presented in Section 3 are both* sound and complete *given a sound and complete constraint solver for the decidable theory $\mathcal{T}$ such that $st(P) \subset \mathcal{T}$. In this case, a directed search with this path constraint generation and constraint solver will thus exercise all feasible program paths exactly once.*

Proof: (sketch) Consider any program path w and its corresponding path constraint pc_w. Since program P is well-formed, every symbolic pointer dereference $*e$ points to a value stored in some valid memory region. In that case, all possible memory regions e may possibly point to are determined using the algorithm of Section 3.2, and results in bound constraints for those regions as described in Section 3.3. Then, given a sound and complete solver for $\mathcal{T}$ such that $st(P) \subset \mathcal{T}$, the path constraint pc_w is satisfiable assuming the bound constraints *if and only if* there is an input assignment that exercises the path w. □

Note that, if a memory-region overflow or underflow is possible along path w, it will be detected provided that (1) every concrete pointer dereference is checked for out-of-bound access violation, *and* (2) every symbolic memory dereference is preceded in the path constraint pc_w by a bound constraint whose negation is checked. As shown in [8], both checks (1) and (2) are actually necessary (i.e., one does not subsume the other). However, in presence of memory-access violations, the program is no longer "well-formed" and the theorem does not hold: although memory-access violations may (and will typically) be detected, it is no longer guaranteed that all feasible program paths will eventually be exercised.

Our memory model assumes that all valid memory regions are input-independent, i.e., that the *address* and the *size* of every memory region are "concrete". In a security setting where program inputs are viewed as controlled by an attacker, assuming that all valid memory regions have a concrete base *address* is equivalent to assuming memory allocation (such as calls to `malloc()`) is not controllable by the attacker, which is often a realistic assumption. With this assumption, our memory model cannot be used to generate test inputs to violate assertions like `assert(&a > &b)` where `&a` and `&b` are base addresses of dynamically-allocated memory regions. But a program that would include such an assertion would not be deterministic, in the sense that a same fixed input could lead to different program executions depending on whether the assertion is violated or not. Nondeterministic programs are not within the scope of Theorem 2.

A more practical limitation is our assumption that valid memory regions cannot have symbolic *sizes*. Indeed, the amount of memory allocated by a program can sometimes depend on some program input or the number of those inputs. However, symbolic reasoning about memory allocation with symbolic sizes is largely orthogonal to reasoning about symbolic address dereferences, which is the main focus of our paper, and this other problem can be addressed using other techniques [16].

5. RELATED WORK AND DISCUSSION

The memory model defined in this paper generalizes and extends previously published ones [13, 3] in the context of dynamic test generation. The simple model in [13] only handles symbolic pointer equalities and inequalities, but does not support pointer arithmetic, multi-dereferences or symbolic writes, and cannot be used to find the assertion violations in the examples of Figures 1 and 6.

The memory model of [3] is more precise and handles both pointer arithmetic and symbolic pointer single dereferences. But it does not handle symbolic pointer multi-dereferences and does not discuss how to deal with symbolic write operations. Although it can deal with the example of Figure 1, it cannot handle the example of Figure 6. Moreover, [3] does not discuss when (i.e., for which programs) its memory model is sound and/or complete. In contrast, the previous section precisely states when our memory model is sound and complete, that is, it formalizes under which assumptions our memory model is the most precise there is. Note that our implementation does not use the optimizations and iterative constraint-refinement scheme discussed in [3] for translating array constraints into SAT constraints as we use a SMT solver (namely Z3) which directly supports array theories.

Memory models used in traditional static program analysis [5, 12] are *abstractions* of the stack and heap of the analyzed program. In contrast, our memory model is very precise and is used in conjunction with a symbolic execution of a specific program path and concrete execution.

Memory models used in static analysis of programs written in higher-level languages often exploit type information to restrict to well-typed objects the set of objects a pointer can point to. This prevents pointers to objects of different types to ever alias each other, and thus facilitates the analysis. Recently [14] and concurrently/independently with our work, such a strongly-typed view of memory and pointers has been adapted to the context of the dynamic unit-test generation tool Pex [11] which performs symbolic execution of .NET programs at the MSIL typed-assembly level (while also supporting memory allocation with symbolic base addresses). In contrast, SAGE performs symbolic execution at the x86 binary level, and our memory model is completely untyped. This way, we support arbitrary pointer casting operations and our analysis does not rely on types, which cannot be trusted when checking for security property violations, i.e., the specific focus of SAGE [9].

Recent work [4, 15, 1] discusses how to perform bit-precise reasoning using static analysis and symbolic execution. The idea is to generate a single SAT formula (possibly refined iteratively) representing the entire program, i.e., all the possible program executions at once. This approach requires models for all system calls and external libraries. Existing implementations often have a strongly-typed view of memory and do not support arbitrary pointer casting operations. Moreover, whole-program formula encodings typically become unsound and/or incomplete in the presence of loops and recursion. Scalability is also problematic and bit-precise analysis is often only intraprocedural, not interprocedural. In contrast, dynamic test generation considers whole program executions one by one, which is expensive, but its (interprocedural) symbolic execution can handle arbitrary system calls and libraries by using testing and concrete values as a fallback. This way, all bugs found are guaranteed to be sound (no false alarms), although full path coverage cannot typically be achieved for large realistic applications and bugs may be missed. However, in practice, we often have no alternative, as the complexity and size of applications like those considered in the next section are beyond the scope of applicability of current automatic interprocedural bit-precise static analysis tools.

Very precise memory models using array theories and select/store operations have also been used for static program verification [2]. There, verification is done compositionally, but requires significant user assistance, in the form of expressive program annotations specifying invariants for every loop and pre/post conditions for every component. In contrast, our memory model is similar but is used in conjunction with a fully-automatic symbolic execution of a single specific program path and concrete execution at a time, without requiring any user annotations.

During execution, valid memory regions need to be tracked by instrumenting dynamic memory allocation. Fairly efficient solutions to this problem are well-known. In SAGE,

Benchmark	GIF		PDF		ANI		Crypto		ZIP	
Mode	Old	New	Old	New	Old	New	Old	New	Old	New
Total Time (s)	185	190	1,060	1,190	80	80	310	315	420	420
Symb. Exec. Time (s)	35	35	190	220	11	10	32	100	35	35
Solver Time (s)	15	17	160	180	2	2	8	65	1.5	1.5
# Tests Generated	54	54	1,124	1,251	129	129	193	167	147	147
# Constraints	499	541	1,782	1,928	318	318	350	352	272	272
Initial Coverage	83,000	83,000	41,760	41,760	19,100	19,100	73,100	73,100	131,200	131,200
Gen1 Coverage	89,500	89,700	45,870	45,940	20,300	20,300	76,000	75,800	133,200	133,200

Figure 8: Microbenchmark statistics.

this is done using TruScan [10].

Our precise memory model may sound expensive. But note that this is very much a "pay as you go" model: for instance, if a program never performs any symbolic pointer dereference, no constraints like those discussed in this paper will ever be generated. The next section presents results of experiments that shed light on this cost/precision tradeoff.

6. EXPERIMENTS

This section presents results of preliminary experiments with our SAGE implementation of the precise memory model introduced in this paper. The goal of these experiments is to estimate the benefit and cost of this memory model. Specifically, we evaluate two configurations: (Old) no symbolic pointer handling and (New) symbolic read operations with single and multi pointer dereferences.

Experiments were performed with the following applications: a GIF image processor, a PDF reader, an animated icons (ANI) parser, a cryptographic (Crypto) certificate processor, and a ZIP file decompressor. For each application, the symbolic inputs are defined as the contents of a single input file. We used the single generation (Gen1) mode of SAGE in which the target application is executed concretely and symbolically on one well-formed seed input file, all the constraints in the path constraint for that execution are negated and solved one-by-one in conjunction with the corresponding prefix of the path constraint (see [9]), and all the resulting new test cases obtained from each satisfiable constraint are executed concretely to measure code coverage and see whether they exhibit a bug. We measured the total running time of each SAGE session including testing the new generated test inputs, the time it took to perform the single symbolic execution, the time taken by the constraint solver, the number of constraints that were successfully negated and solved to yield a new test case, the total number of constraints in the path constraint generated by the symbolic execution, the (rounded) initial instruction coverage obtained by running the target program with the seed input, and the (rounded) total (Gen1) instruction coverage obtained after also running the target program with all the new test inputs generated.

Results are summarized in Figure 8. The ANI and ZIP programs do not use any input in any indirect memory accesses. As a result, there were no symbolic dereference constraints: the new memory model did not have any effect on these programs. The new GIF run generated more constraints than the old run, but the same number of constraints were successfully inverted in both runs. Apparently, some of the constraints that were inverted in the old run become unsatisfiable and are not inverted in the new run due to increased precision. As a result SAGE generates fewer divergences and the overall quality of the generated tests is better in the new run as witnessed by the marginally better coverage produced by the new run. Note that the symbolic execution and constraint solving times are not affected by the overhead of the new memory model in those three experiments.

The new PDF run generates more constraints and more tests. Although overall coverage is a little better as a result, this comes at a cost: the running time is slightly higher both because there are more tests to evaluate and because of the overhead in symbolic execution and constraint solving.

In the Crypto example, we see that although almost no additional constraints are added by the new run, a significant number of constraints are no longer satisfiable because they contain additional symbolic pointer expressions. Although symbolic execution and constraint solving incur an overhead with the new memory model, this time is made up overall because there are fewer new tests to run and evaluate. The marginal decrease in coverage for the new run is caused by divergences arising from symbolic-execution imprecision that is unrelated to the memory model. Indeed, increased precision in symbolic pointer reasoning can discover more inter-connections between individual constraints, but solutions to larger constraints tend to set new values to more inputs. Perturbing more inputs may then cause more divergences because of other hidden constraints not captured by symbolic execution.

Overall, these results are promising in that we do not observe a drastic total time increase as a result of introducing the new memory model. Therefore, even a small chance of finding a new bug thanks to increased precision justifies the memory model presented here.

We also performed longer 24-hours searches with the GIF and Crypto benchmarks, but those did not uncover new bugs in those programs.

Over the past few months, our new memory model has been deployed and used on a large scale in a fuzzing lab with hundreds of dedicated machines running SAGE for weeks at a time, each on a different Windows application. These longer searches were able to discover new bugs in several of those applications, but it is hard to attribute those results to the new memory model alone. Indeed, SAGE is a complex system with many features and parameters influencing its effectiveness, and it is hard to single out the effect of any single feature, like our new memory model, in the "production testing" fuzzing lab owned by the Microsoft Windows organization, where research experiments and precise data collection are not possible due to various reasons.

```
BYTE ** __cdecl decode_packets(char *buffer)
{
   // input format is <byte:number of packets>(<byte:packet#><PACKET_SIZE bytes:content of packet#>)*
   // packet# ranges from 0 to MAX_PACKET, each packet contains PACKET_SIZE bytes
   // packets can be out of order !

   int i,j;

   BYTE ** multi_array = new BYTE*[MAX_PACKET];
   for (i=0; i<MAX_PACKET; i++)
      multi_array[i] = new BYTE[PACKET_SIZE];

   for (i=0;i<MAX_PACKET;i++)
      for (j=0;j<PACKET_SIZE;j++)
         multi_array[i][j]=0;

   // decode packets

   int number_of_packets;
   int packet_id;

   number_of_packets = (int)buffer[0];
   if ((number_of_packets > MAX_PACKET) || (number_of_packets < 0))
      return;
   for (i=0;i<number_of_packets;i++){
      packet_id = (int)buffer[(i*(PACKET_SIZE+1))+1];
      if ((packet_id >= MAX_PACKET) || (packet_id < 0)) // validate packet_id (*)
         return; // abort since input buffer is corrupted
      for (j=0;j<PACKET_SIZE;j++)
         multi_array[packet_id][j] = buffer[(i*(PACKET_SIZE+1))+j+2]; // (**)
   }

   // ...

   if ((number_of_packets<MAX_PACKET) && (multi_array[number_of_packets][0] != 0)) // (***)
      assert(false); // can be violated!

   // ...
}
```

Figure 9: An example of packet decoder.

7. EXAMPLE: A PACKET DECODER

In this section, we present a simplified version of an actual C++ packet decoder as an example to illustrate when the new memory model described in this paper is necessary to detect an error with systematic dynamic test generation.

Figure 9 presents a program that takes as input a `buffer` containing a sequence of bytes storing a network message. Each message is supposed to start with one byte declaring how many packets are encoded in the message, followed by a finite (and bounded) sequence of packets, each starting with a one byte packet identifier (number) followed by a fixed number `PACKET_SIZE` of bytes representing the packet content.

For instance, with a `PACKET_SIZE` of 4 and a `MAX_PACKET` of 10, the input `buffer` has a maximum size of

`1+((1+PACKET_SIZE)*MAX_PACKET)`

i.e., 51 bytes, and the following network message

`03 00 A B C D 01 E F G H 02 I J K L`

is well-formed: the first byte indicates there are 03 packets in this message, packet 00 whose content is ABCD, packet 01 whose content is EFGH, and packet 02 with content IJKL.

The function `decode_packets` is invoked on the arrival of each network message, parses the message, stores every packet identifier and their corresponding packet content in a data structure `multi_array`, and then returns a pointer to that data structure (code not shown here). The code contains an assertion in line (***) that tests as a sanity check (among others) whether, whenever the number of packets `number_of_packets` is less than the maximum allowed number `MAX_PACKET`, the content of the packet number `number_of_packets` in the data structure is still 0.

Starting with a well-formed input network message like the one above, previous systematic dynamic test generation techniques and tools [7, 13, 3, 9] are unable to find another network message leading to a violation of that assertion at line(***): if given enough time and resources, they would eventually stop without reporting any error after having exhausted all the possible path constraints they can generate, each corresponding to a different unfolding of the main loop `for (i=0;i<number_of_packets;i++)` (since `number_of_packets` is a symbolic input) combined with tests in line (*) (since `packet_id` is a symbolic input). Note that the exact number of path constraints depends on the value of the constants `MAX_PACKET` and `PACKET_SIZE`, but the outcome is always the same: no error is detected since modifying the values of these constants does not help in finding the assertion violation in this example.

In contrast, from the very first run of this program with the well-formed input above, dynamic test generation using our new memory model with both symbolic writes (line **) and multi-pointer dereferences (line ***) is able to generate, at the first visit of line (***), a symbolic constraint checking whether one of the packets can have a packet identifier equal to `number_of_packets` and a first non-null byte as its packet content. This constraint is solvable, and one of the solutions corresponds to the new test input

```
03 00 A B C D 01 E F G H 03 I J K L
```

Running this program with that modified network message as input indeed leads to the assertion (***) being violated.

Note that one way to fix this error is to replace `(packet_id >= MAX_PACKET)` in line (*) by `(packet_id >= number_of_packets)`.

8. CONCLUSIONS

This paper presents a memory model for precise pointer reasoning in dynamic test generation. Although more complex than closely related work and nontrivial to implement, our new memory model is still conceptually simple. Given any symbolic pointer, we determine the valid memory region its concrete address points to. If the pointer points to no such region, we report a memory-safety violation. Otherwise, we force the symbolic address to be confined within this valid memory region and take a snapshot of the content of that region; separately, we also check whether the symbolic address could possibly overflow or underflow the valid memory region. For chains of symbolic pointer dereferences, the model is extended to sets of memory regions.

Compared to memory models used for static program analysis, our memory model is unique in that it leverages concrete values available at run-time, an approach that would not be possible with static analysis. This feature is key to achieve high precision. Another contribution of this paper is the maximum precision theorem: the memory model we develop is the most precise there is, under the assumptions specified in Section 4. This result settles the discussion of precise memory models for pointer reasoning in the context of dynamic test generation under those assumptions.

Overall, dynamic test generation, and testing in general, are of course complementary to other defect detection techniques, like static program analysis. In practice, static analysis of large programs is less computationally expensive, but is usually both unsound (may generate false alarms) and incomplete (may miss bugs) from a bug-finding perspective. In contrast, dynamic test generation for large programs is expensive and typically incomplete (may miss bugs), but it is precise and sound from a bug-finding perspective: each bug found can be reproduced with a concrete execution trace. This property is key to the rapidly increasing popularity of dynamic test generation, which is now used daily at Microsoft for checking security properties of software, in addition to static analysis and blackbox fuzz testing.

Acknowledgments. We thank Mike Barnett, Leonardo de Moura, Rustan Leino and Nikolai Tillmann for helpful comments. We also thank the anonymous reviewers for their suggestions to improve the presentation.

9. REFERENCES

[1] D. Babic and A. J. Hu. Structural Abstraction of Software Verification Conditions. In *Proceedings of CAV'2007 (19th Conference on Computer Aided Verification)*, Berlin, July 2007.

[2] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO'2005 (4th International Symposium on Formal Methods for Components and Objects)*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.

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

[4] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of TACAS'2004 (Tools and Algorithms for the Construction and Analysis of Systems)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, January 1977.

[6] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS '08: Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[7] 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.

[8] P. Godefroid, M.Y. Levin, and D. Molnar. Active Property Checking. In *Proceedings of EMSOFT'2008 (8th Annual ACM & IEEE Conference on Embedded Software)*, pages 207–216, Atlanta, October 2008. ACM Press.

[9] P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, San Diego, February 2008.

[10] 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.

[11] Pex. Web page: `http://research.microsoft.com/Pex`.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg, Florida, January 1996. ACM Press.

[13] 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.

[14] D. Vanoverberghe, N. Tillmann, and F. Piessens. Test Input Generation for Programs with Pointers. In *Proceedings of TACAS'2009 (15th Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, York, April 2009.

[15] Y. Xie and A. Aiken. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of POPL'2005*, 2005.

[16] R. Xu, , P. Godefroid, and R. Majumdar. Testing for Buffer Overflows with Length Abstraction. In *Proceedings of ISSTA'08 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 27–38, Seattle, July 2008.