Test Input Generation for Programs with Pointers

Dries Vanoverberghe[1,*], Nikolai Tillmann[2], and Frank Piessens[1]

[1] Katholieke Universiteit Leuven, Belgium
{Dries.Vanoverberghe,Frank.Piessens}@cs.kuleuven.be
[2] Microsoft Research, Redmond, USA
nikolait@microsoft.com

Abstract. Software testing is an essential process to improve software quality in practice. Researchers have proposed several techniques to automate parts of this process. In particular, symbolic execution can be used to automatically generate a set of test inputs that achieves high code coverage.

However, most state-of-the-art symbolic execution approaches cannot directly handle programs whose inputs are pointers, as is often the case for C programs. Automatically generating test inputs for pointer manipulating code such as a linked list or balanced tree implementation remains a challenge. Eagerly enumerating all possible heap shapes forfeits the advantages of symbolic execution. Alternatively, for a tester, writing assumptions to express the disjointness of memory regions addressed by input pointers is a tedious and labor-intensive task.

This paper proposes a novel solution for this problem: by exploiting type information, disjointness constraints that characterize permissible configurations of typed pointers in byte-addressable memory can be automatically generated. As a result, the constraint solver can automatically generate relevant heap shapes for the program under test. We report on our experience with an implementation of this approach in Pex, a dynamic symbolic execution framework for .NET. We examine two different symbolic representations for typed memory, and we discuss the impact of various optimizations.

Keywords: Test input generation, symbolic execution, pointers.

1 Introduction

Today, testing is still by far the most effective way to improve software quality. Recently, there is a lot of effort to automate different parts of the testing process. One aspect is test input generation for an open program, i.e. a program that takes

[*] Dries Vanoverberghe is a research assistant of the Fund for Scientific Research - Flanders (FWO). Most of the work was conducted while he was visiting Microsoft Research.

S. Kowalewski and A. Philippou (Eds.): TACAS 2009, LNCS 5505, pp. 277–291, 2009.
© Springer-Verlag Berlin Heidelberg 2009

inputs. The challenge is to generate a small set of test inputs that maximizes code coverage.

Symbolic execution[1] is a well known static analysis technique to generate test inputs, where the program is executed with symbolic inputs instead of concrete inputs. A constraint solver is used to compute test inputs for particular execution paths. Since its early introduction, there has been a tremendous increase in computing power which paved the road for engineering more efficient constraint solvers and more precise analysis tools. Not surprisingly, symbolic execution for test input generation has become popular lately (e.g. Java Pathfinder [2], EXE [3], Cute [4], Sage [5], Pex [6], ...). Most of these tools use a variant of symbolic execution where the program is repeatedly executed with concrete inputs. During a concrete execution, the program is monitored to build a symbolic representation of the executed path; the next test inputs are constructed in such a way that they will exercise a new execution path. Constraints that are outside of the scope of the employed constraint solver can be simplified by using observed concrete values instead of symbolic values. This variant has been called DART [7], concolic execution [4], and dynamic symbolic execution[5,6]. The many tools implementing it show its relevance in practice (EXE [3], Cute [4], Sage [5], Pex [6], Yogi [8], Vigilante [9], Bitscope, ...).

In the context of static analysis tools, reasoning about programs with pointers has traditionally been challenging. For test input generation this is not different: Some tools based on dynamic symbolic execution don't support symbolic pointer reasoning and use the concrete value for pointers as an under-approximation instead. For instance, EXE [3] uses concrete values whenever it encounters a double dereference and Sage [5] always uses concrete values. Although this means that the exploration can sometimes be incomplete, for testing tools this is appropriate. If symbolic pointers are supported, simplifying assumptions often make pointer reasoning incomplete (for example, treating pointers as references instead of integers with arbitrary arithmetic [4]).

In this work we focus on test input generation for programs manipulating (complex) data structures with pointers. A common approach is to first assume that an invariant holds for the data structure; then to apply an operation on the data structure; and finally to assert that the invariant still holds. Expressing the invariant for arbitrary pointers is often tedious, as a large portion of the invariant usually states that different pointers point to different memory regions which do not overlap in type-incorrect ways. This is a property that is guaranteed by safe managed languages such as Java and C#, but it is not guaranteed by languages that allow unsafe memory accesses, such as C and C++.

We studied the problem in the context of Pex [6], a dynamic symbolic execution tool for the Common Intermediate Language (CIL) of the .NET Framework. CIL consists of a strongly typed verifiable core, extended with unsafe instructions (e.g. for unsafe memory accesses through pointers) that are available only when the program is sufficiently trusted by the user. By using these unsafe instructions, C programs can be translated to CIL.

The contributions of this paper are the following:

– We propose two different symbolic representations for unsafe memory. The main idea of these representations is to exploit the type information that is present in the CIL to the fullest extent possible.
– We present a number of axioms that characterize that pointers only overlap in type-correct ways.
– We introduce a two-phase solving scheme to improve the efficiency of the solving process by performing multiple incremental queries to the solver.
– We report on an experimental evaluation based on an implementation in Pex.

The remainder of this paper is structured as follows: Section 2 explains more details about dynamic symbolic execution. Section 3 motivates why automatic test input generation for programs with pointers is challenging. Next, we shortly introduce our type system (Section 4) before explaining how we model memory (Section 5) and how we enforce disjointness of input data accessed through typed pointers (Section 6). In Section 7 we discuss how we can improve the performance of the resulting system. Section 8 contains an evaluation of our approach using red-black trees and linked lists as data structure. Finally, we treat related work in Section 9 and conclude.

2 Background: Dynamic Symbolic Execution

Symbolic execution [1] is a technique to explore the behavior of a program under all possible inputs. Instead of using concrete inputs, the program is executed with symbols representing arbitrary values. As a result, the inputs are partitioned into equivalence classes that follow the same execution path. This path is represented by the path-condition, a conjunction of constraints on the symbolic inputs. At each branch during the execution of the program, the inputs are split into two equivalence classes by conjoining the (negated) branch condition with the path-condition. A satisfiability modulo theory (SMT) solver is used to check whether the resulting path-conditions are feasible and to compute a set of inputs that represents that particular execution path.

Theoretically, given a correct (sound and complete) constraint solver and a correct symbolic execution engine, symbolic execution of programs with a finite number of finite paths is equivalent to program verification. In reality, the symbolic execution engine will often be limited in its completeness due to instructions with complex behavior such as floating point operations or code that is outside the scope of the symbolic execution engine such as operating system calls.

Dynamic symbolic execution (also named DART [7] or concolic execution [4]) is a variant of symbolic execution in which the symbolic constraints are gathered by monitoring the program during a concrete execution and maintaining a symbolic representation on the side. First the program is executed with arbitrary inputs. Then, a constraint solver is used to find inputs that drive the

Set $J := false$	*intuitively, J is the set of already...*
loop	*...analyzed program inputs*
Choose program input i such that $\neg J(i)$	*stop if no such i can be found*
Output i	
Execute $P(i)$; record path condition C	*in particular, $C(i)$ holds*
Set $J := J \vee C$	
end loop	

Algorithm 1. Dynamic symbolic execution

execution along new execution paths. The advantage of this approach is that even in the context of an imprecise representation, there will never be false positives. When the concrete execution diverges from the intended execution path it can be detected and reported. Furthermore, when precise symbolic execution is impossible, concrete values can be used to simplify the constraints. In that case, the set of covered execution paths is an underapproximation of the feasible execution paths, which is appropriate for testing.

Algorithm 1 shows the general dynamic symbolic execution algorithm.

For symbolic execution, the constraint solver needs to support a variety of different theories, such as bit-vectors with all common integer arithmetic operations to model machine integers, the theory of arrays with read and write-functions for arrays, and tuples to represent structs. Often, uninterpreted functions are used (e.g. for typing constraints); their possible interpretations are restricted, or fully determined, by introducing background axioms (potentially with quantifiers). Satisfiability modulo theory (SMT) checkers are efficient to reason about such a combination of theories. They handle quantified background axioms by instantiating them when a designated pattern occurs in the formula that is checked for satisfiability. This approach is only complete when the patterns are carefully chosen. Since modern SMT checkers can generate models, i.e. satisfying assignments, they can be used as constraint solvers for symbolic execution. In this work, we use Z3 [10] as the constraint solver.

3 Motivating Example

In the following, we will illustrate the problem of test input generation for programs whose inputs are pointers. Consider the program in Figure 1. The function *test* tests the method *enqueueTail*, a part of the enqueue operation of linked lists. It takes the tail pointer p, a pointer to a freshly initialized node q and a new value *val* as input and adds the value as last element of the list. In the beginning of the test, we assume that p and q are not null and different. Furthermore, the *next* field of both p and q must be null. After the *enqueueTail* operation, the value of the next node of p must equal the new value.

To cover this program, a dynamic test generation tool must generate values for pointers, i.e. memory addresses, and assign values to the memory locations at these addresses. In the first execution, arbitrary values can be assigned to

```
void enqueue(Queue* q, int val) {
  Node* newNode = malloc(sizeof(Node));
  enqueueTail(q -> tail, newNode, val);
  q -> tail = newNode; }
void enqueueTail(Node* p, Node* q, int val) {
  q -> value = val;   p -> next = q; }
void test(Node* p, Node* q, int val) {
  Assume(p != 0 && q != 0 && p != q);
  Assume(p -> next == 0 && q -> next == 0);
  enqueueTail(p, q, val);
  Assert(p -> next -> value == val); }

struct Node {
  int value;
  Node* next;
}
struct Queue {
  Node* head;
  Node* tail;
}
```

Fig. 1. A motivating example based on linked lists

the inputs; we choose the value 0 for pointers, and 0 for integers. During the execution, a symbolic representation is maintained, in which the pointers are treated as regular integers. (We use this representation of pointers as it reflects the encoding of pointer operations at the level of the execution machine, where all high-level type information has been erased.) In the first execution, p is null, so the path-condition will be $p == 0$. To explore new behavior, we need to find a value for p such that $p! = 0$. The constraints will be solved incrementally, and the final constraint to pass the first assumption is $p \mathrel{!}= 0 \;\&\&\; q \mathrel{!}= 0 \;\&\&\; p \mathrel{!}= q$. The constraint solver might find a solution such as $p = 1, q = 2$, where 1 and 2 are integers that represent the addresses of the second and third byte of the addressable memory. During the next execution, this will likely result in an access violation when trying to read the value of the next field of p, since the program does not have access to these locations when it is executed within a regular process of a typical operating system. Of course, this technical issue can be solved by allocating a big chunk of memory before any test is executed, and adding additional constraints on p and q to express that these pointers must be in that memory region.

Next, in order to pass the second assumption, we again send a set of constraints to the constraint solver. This time, the solver will not only give a model for the inputs, but also for some values in memory. Our tool parses this model and initializes the memory with the supplied values before executing the test. For example, in this case it would assign null to the *next* fields of p and q. Although this is not really challenging, integration testing tools (like Sage [5] and EXE [3]) often don't support this. For unit testing tools, dealing with pointers as input is essential.

Besides these technical issues, the dynamic symbolic execution process will report an assertion violation. This is because the program fails to specify the assumption that the memory regions, to which p and q point, do not overlap (are disjoint). Indeed, if for instance $q = p + 4$ then the next field of p has the same address as the value field of q. Since the next field of p is updated after the value of q is set to val, the value of q is destroyed. Therefore, the value of the next node of p is not equal to val.

In practice, it is surprisingly tedious to express all disjointness assumptions. Especially for complicated data structures, like trees or graphs, where the number of disjointness constraints is quadratic in the number of nodes. Forgetting one assumption can lead to a test case that is particularly hard to debug. Furthermore, our experience shows that developers are likely to forget such assumptions as they are often taken for granted.

In this paper, we seek to exploit the information present in the type system, and the signature of the test function. We restrict input generation for pointers in such a way that pointers are always used in a type-correct way. As a result, it is no longer necessary to encode this quadratic amount of disjointness constraints manually. Instead, the type information is encoded in the constraints for the constraint solver.

4 Types

In this section, we give a short sketch of the type system that we considered – a small subset of the type system of full CIL that we found relevant for unsafe memory operations. Figure 2 shows the syntax of types in our system. A type can either be a primitive type or a struct type. $I1$ is a byte, $I2$ a two-byte entity, and so on. $R4$ represents a four-byte float, and so on. Struct types are essentially a list of types with labels. In this way, types are basically trees where the leaves are primitive types and the nodes are struct types. Depending on the architecture of the machine, pointers would be represented as $I4$, or $I8$. We will use $I8$ in the following.

$$\begin{aligned} \mathit{Type} &:= \mathit{PrimitiveType} \mid \mathit{StructType} \\ \mathit{PrimitiveType} &:= I1 \mid I2 \mid I4 \mid I8 \mid R4 \mid R8 \mid \ldots \\ \mathit{StructType} &:= \mathit{Type}\ \mathit{Label};\ \mathit{StructType} \mid \mathit{Type}\ \mathit{Label} \end{aligned}$$

Fig. 2. Syntax types

We do not consider empty structs. We assume the presence of the function $sizeof$ that returns the size of a type in memory in bytes, and the function $nextOffset$ that gives the offset of the second label of a struct in bytes. The label represents a named field of a struct. The functions $nestedIn$ and $unnested$ can be used to test whether one type is nested in another type or not.

5 Memory Representations

This section describes how memory is modeled to support precise test input generation for unsafe pointers. From the point of view of the concrete execution engine, memory is just one big byte array. Whenever a value at a particular offset is read, the execution engine reads a number of consecutive bytes from memory and converts them to a value of the requested type.

The most precise way to model memory symbolically is to stay as close to the concrete semantics as possible. Logically, pointers can be encoded as regular integers, and memory can be represented as a map from integers to bytes. Reads and writes are represented as selecting from this map or updating it at a number of consecutive indices. This precision comes with a cost for every memory access with a type bigger than a byte. To reduce this cost, we exploit the typing information to simplify the memory representation. Since well-typed pointers can not overlap, we can split the memory into different maps according to the type. We explored two different variants of this scheme:

map per type. For each type T, we keep a map MM_T from integers to values of type T. For struct types, this means that the value of the field can be retrieved in two ways. For example, assume we have a struct T with a field of type S: First, we can select the entire struct of type T from the MM_T and get the value of its field. Alternatively, we can compute the address of the field and select the value directly from MM_S.

To keep the different maps consistent, we introduce an axiom over the maps that relates the initial memory maps. Whenever we write a complete struct to a pointer, we update both the memory map of the struct and the memory maps of its fields recursively. Furthermore, we use typing information that is present in CIL to reverse engineer when an assignment to a pointer is an assignment to a field of a struct. In that case, we do not only update the field, but also the struct. Unfortunately, when complex pointer manipulation operations are performed, it might not always be statically known that a pointer points to a field of a struct. In this case, the symbolic representation is imprecise with respect to the real representation.

map per primitive type. We only maintain maps for primitive types. Reading or writing complete structs is done recursively over all fields. This representation is no longer imprecise since the constraint solver now reasons about the relation between different pointers.

6 Enforcing Disjointness

As mentioned before, the memory representations introduced in Section 5 are only precise under the assumption that pointers separate memory into disjoints memory regions that are only accessed according to one particular type (or compatible types in the case of nested structs). In this section we explain how we use the typing information to enforce this assumption.

To encode the typing information for the constraint solver, we want to express that a pointer p has type T (e.g. $typeOf(p) == T$). Because our type system contains structs, it is possible though that one pointer has multiple types. Consider a pointer to a struct T whose first field has type S. This pointer has both T and S as type. To this end, we could introduce a relation $typed(T, p)$ to express that a pointer p has type T.

However, using such a relation as a basic block of our definitions would be inefficient: We would have to create constraints to forbid all illegal combinations

of types, e.g. to indicate that a $typed(byte, p)$ and $typed(int, p)$ is mutually exclusive. To achieve better performance, we stratify types according to their type level Essentially, the type level is the largest nesting depth. If we conceptually think of a type as a tree, the type level is the height of the tree.

For each type level, we define an uninterpreted function $typeOf_{typeLevel}$ that takes a pointer as input and returns a value representing the type of the pointer. Now we can define $typed(T, p)$ as syntactic sugar for $typeOf_{typeLevel(T)}(p) == typeConstant(T)$. Since the $typeOf$ symbols are functions, the theory of equality over uninterpreted function symbols can infer that $typeOf_l(p)! = typeOf_l(q)$ implies that p and q are different. Using only the predicate $typed$, these implications would have to be encoded as quantifiers, which is potentially less efficient.

The semantics of these uninterpreted functions is given by the axioms defined in Figure 3. Whenever a pointer to a struct type is typed, then the pointers to the fields of this struct are also typed according to their type. Furthermore, pointers that are typed must always be in a predefined region of memory that we allocated for this purpose. The constants $vmbase$ and $vmsize$ represent the base address and size of this region.

$$
\begin{aligned}
&\forall\ (T : Type,\ t : label,\ S : StructType,\ p : I8),\ \underline{typed((T\ t; S),\ p)}\\
&\quad => \ typed(T,\ p)\ \&\&\ typed(S,\ (p + nextOffset(T\ t; S)))\\
&\forall\ (T : Type,\ t : label,\ p : I8),\ \underline{typed((T\ t),\ p)}\ => \ typed(T,\ p)\\
&\forall\ (T : Type,\ p : I8),\ \underline{typed(T,\ p)}\\
&\quad => \ p \geq vmbase\ \&\&\ p + sizeof(T) \leq vmbase + vmsize\ \&\&\ p + sizeof(T) > p
\end{aligned}
$$

Fig. 3. Axioms over *typed* function

Figure 4 shows the disjointness axioms that apply to pointers and Figure 5 introduces a number of helper functions.

First, two pointers with same type must either be equal or they do not overlap. Second, two pointers with different types where neither of the types is nested inside the other type never overlap. Finally, when one type is nested in another type, a pointer to the nested type can either be correctly embedded with respect to the pointer of the other type or both pointers do not overlap. Correctly embedded means that if the nested type is equal to a field type, then the embedded pointer can be equal to this pointer. Alternatively, if the nested type is nested inside a field type, then the embedded pointer can be embedded in the pointer to the field. Together with the axiom to propagate type information to the fields of a struct, these three axioms precisely define how pointers can relate to each other.

In Section 2, we mentioned that SMT solvers need a carefully designed (set of) patterns to instantiate quantifiers. In Figure 3 and 4, these patterns are illustrated by underlining them. We do not provide patterns for the first quantifier in the last two disjointness axioms because they are statically expanded. Two patterns in one quantifier represent one multi-pattern rather than two patterns. The patterns in our axioms only occur on the left hand side of an implication, and the right hand side will only generate the same pattern with terms that are structurally smaller. This corresponds to defining a function by recursion

$$\forall\ (T : Type, p1 : I8,\ p2 : I8),\ \underline{typed(T,\ p1)}\ \&\&\ \underline{typed(T,\ p2)}\ =>$$
$$\quad p1 == p2 \parallel noOverlap(T,\ T,\ p1,\ p2)$$
$$\forall\ (T1 : Type,\ T2 : Type),\ unnested(T1,\ T2)\ =>$$
$$\quad \forall\ (p1 : I8,\ p2 : I8),\ \underline{typed(T1,\ p1)}\ \&\&\ \underline{typed(T2,\ p2)}\ =>$$
$$\quad noOverlap(T1,\ T2,\ p1,\ p2)$$
$$\forall\ (T1 : Type,\ T2 : Type),\ nestedIn(T1,\ T2)\ =>$$
$$\quad \forall\ (p1 : I8,\ p2 : I8),\ \underline{typed(T1,\ p1)}\ \&\&\ \underline{typed(T2,\ p2)}\ =>$$
$$\quad noOverlap(T1,\ T2,\ p1,\ p2) \parallel correctlyEmbedded(T1,\ T2,\ p1,\ p2)$$

Fig. 4. Disjointness axioms

$$noOverlap(T1 : Type,\ T2 : Type,\ p1 : I8,\ p2 : I8)\ :=$$
$$\quad p1 \geq p2 + sizeof(T2)\ \parallel\ p2 \geq p1 + sizeof(T1)$$
$$embedded(T2 : Type,\ p1 : I8,\ p2 : I8)\ :=\quad p1 \geq p2\ \&\&\ p1 < p2 + sizeof(T2)$$
$$correctlyEmbedded(T1 : Type,\ T2 : Type,\ p1 : I8,\ p2 : I8)\ :=$$
$$\quad \textbf{match}\ T2\ \textbf{with}$$
$$\quad |\ PrimitiveType\ =>\ T1 == T2\ \&\&\ p1 == p2$$
$$\quad |\ StructType\ =>$$
$$\qquad \textbf{match}\ StructType\ \textbf{with}$$
$$\qquad |\ T\ t;\ StructType'\ => embeddedInField(T1, T, p1, p2) \parallel$$
$$\qquad\quad correctlyEmbedded(T1, StructType', p1, p2 + nextOffset(StructType))$$
$$\qquad |\ T\ t\ => embeddedInField(T1, T, p1, p2)$$
$$\qquad \textbf{end}$$
$$\quad \textbf{end}$$
$$embeddedInField(T1 : Type,\ T2 : Type,\ p1 : I8,\ p2 : I8)\ :=$$
$$\quad \textbf{match}\ T2\ \textbf{with}$$
$$\quad |\ PrimitiveType\ =>\ T1 == T2\ \&\&\ p1 == p2$$
$$\quad |\ StructType\ =>$$
$$\qquad nestedIn(T1,\ StructType)\ \&\&\ embedded(StructType, p1, p2)$$
$$\quad \textbf{end}$$

Fig. 5. Helper functions for disjointness axioms

over the structure of arguments. Therefore, there will only be a finite number of instantiations, and the use of pattern based instantiation is complete for our axioms.

Together with the disjointness axiom, the memory representations of Section 5 are precise for all well-typed programs. Furhtermore, incorrect pointer arithmetic can be detected by automatically checking whether a pointer has a compatible type prior to every memory access.

7 Optimizations

7.1 Two-Phase Solving

After initial experiments, we observed that some of the constraint systems generated during our symbolic execution are particularly hard for the constraint

solver. Furthermore, we noticed that the solver mainly had a hard time when the constraints are unsatisfiable. When they are satisfiable, the constraint solver usually gives a solution fairly quickly.

This can be explained by analyzing the disjointness axioms. First of all, the axioms cause a quadratic number of disjointness constraints in the amount of pointers. Furthermore, each disjointness constraint is actually a disjunction of two inequalities. To make matters even worse, pointers are represented as bitvectors, therefore an inequality causes the creation of a circuit, i.e. a representation of the inequality by logical gates operating at the bit-level. Not surprisingly, the constraint systems give rise to a large number of case splits, especially when all, or at least many, cases have to be enumerated which often happens when the constraints are unsatisfiable.

To improve the performance, we split constraint solving in two phases:

- First, we perform a satisfiability check for a simplified set of constraints. In particular, all disjointness constraints are replaced by a single disequality between the two pointers (E.g. $noOverlap(T1,\ T2,\ p1,\ p2) \rightarrow p1\ != p2$). The resulting constraint system is weaker than the original one. If the satisfiability check fails, there will not be a solution for the original constraint system and we can skip the second phase.
- Then, we exploit the incremental nature of the constraint solver and add the full disjointness axioms to the simplified constraint system. In principle, checking the full constraints first and only adding them when necessary is possible, but we did not implement such a scheme. In any case, the full constraints are necessary to remain precise.

7.2 Alignment

Data is said to be aligned when its address is divisible by certain powers of two. For example, on the X86 architecture, a 4-byte (8-byte) entity is aligned when its address is divisible by four (eight). Accessing misaligned data often imposes a performance penalty; it may even be forbidden. As a result, most compilers automatically align data structures according to their type by inserting padding bytes.

With alignment, two pointers with a primitive type will always be equal or not overlapping. For primitive types, we can exploit alignment by replacing the disjointness axiom for pointers of the same type by an alignment constraint (which states that the lower bits are equal to zero). The advantage is that the alignment constraint does not cause a quadratic number of pointer inequalities.

8 Evaluation

Since our approach aims at generating inputs for pointers as well, we evaluate it in the context of data structures where pointer reasoning is essential.

First, we test an implementation of red-black trees, a self-balancing binary search tree, taken from the Windows source code base. This is a challenging test

case because red black trees have complicated constraints over a data structure with pointers. For example, the sum of all black nodes on a path from the root to any leaf node is always the same. Furthermore, in this performance-optimized implementation the leafs of the red-black tree are represented by a sentinel node which is nested inside the tree itself. This was an excellent test to see if nested types are working correctly.

We have tested the red-black tree with both memory representations described in Section 5. To evaluate the overhead of the enforcement of the disjointness conditions, we manually created a version of the test case with the extra constraints that all pointers are aligned modulo 1024, which can be expressed efficiently by operations at the bit-level. As a consequence, different pointers never overlap. This trick was much more convenient to enforce the disjointness of the pointers than manually walking over the entire data structure and enforcing the disjointness constraints. Furthermore, these logical conditions are slightly more efficient than encoding the disjointnesses manually. Therefore, comparing the performance of the disjointness with this system is not completely fair, but it already gives a good idea whether our system is much slower because of the disjointness axioms or not. This technique has limited potential for automatisation because pointers to nested structs are not necessarily aligned.

When changing the constraint systems, the order in which execution paths are explored usually changes, since the next execution path depends on the test inputs computed by the constraint solver. In order to compare the performance of the different memory representations, we inserted a subtle bug in the fixup routine of the red-black trees. We stop the test input generation as soon as the first two test cases that trigger the bug have been reported.

The results can be seen in Figure 6. On the Y-axis, we report the number of tests that has been generated. The memory representation with a map per type clearly outperforms the representation with a map per primitive type. In Section 5, we mentioned that the first representation is potentially imprecise when it is impossible to statically know if an assignment to a pointer is an assignment to a field of a struct. In practice, this imprecision never occured while executing the red-black tree benchmark. A second observation is that the disjointness constraints do not have a big impact in the first memory representation (16s vs 28s). For the second representation though, it deteriorates the performance severely.

To test our optimizations, we have executed the red-black tree again using the first memory representation and all combinations of the optimizations. We again stopped the input generation when we found the first two failing test cases. The result can be seen in Figure 7. With two-phase solving, the test input generation is clearly faster. In combination with two-phase solving, alignment does not seem to offer too much improvement. Without two phase solving, alignment seems to deteriorate the performance, although this is hard to explain. In addition, we computed the average time for the constraint solver to handle a query. Without two-phase solving, the average is $.13s$. With two-phase solving, the averages for the first and the second phase are $.02s$ and $.05s$ respectively. These number

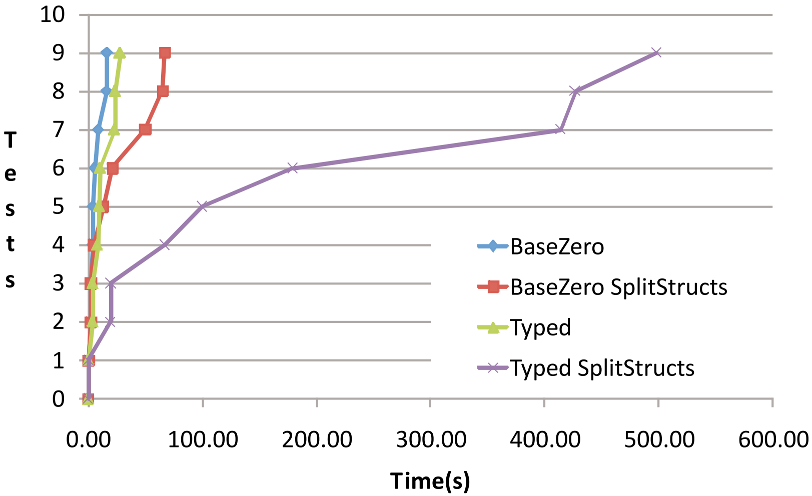

Fig. 6. Red black tree: Memory representations (*BaseZero* uses alignment modulo 1024 to enforce disjointness and *Typed* uses the the disjointness axioms. *SplitStructs* means that we only maintain a map per primitive type.)

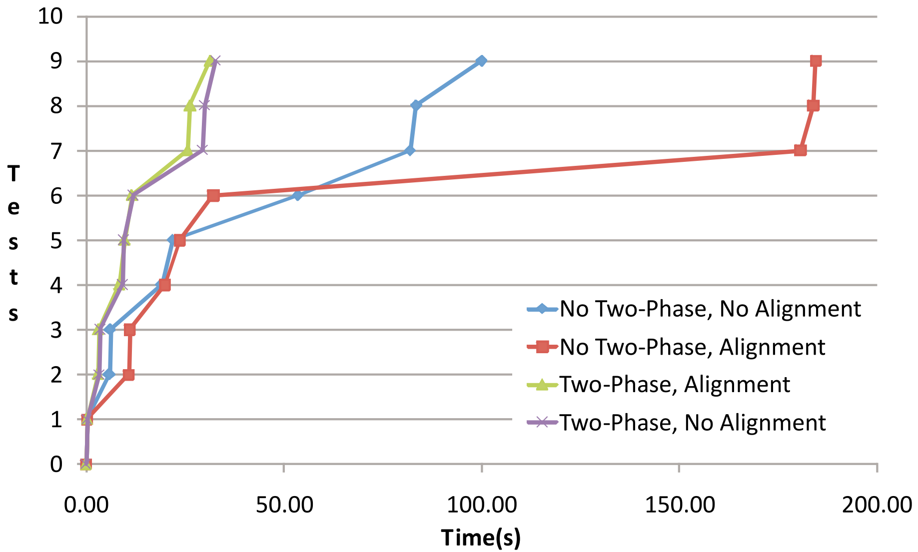

Fig. 7. Red black tree: Optimizations

confirm our hypothesis that we can improve the performance by doing a fast satisfiability check first.

Finally, we tested all combinations of the different options (memory representation, two-phase solving and alignment) on a linked list data structure. We used a fixed timeout of $600s$, and we report the size of the largest queue that has been

found over time. Figure 8 shows the results. Two-phase solving has the biggest impact on the size of the linked lists that are being generated. With two phase solving, the first memory representation seems to perform better. Without two phase solving, the second memory representation seems better. Also, alignment seems to improve the performance in three out of four cases.

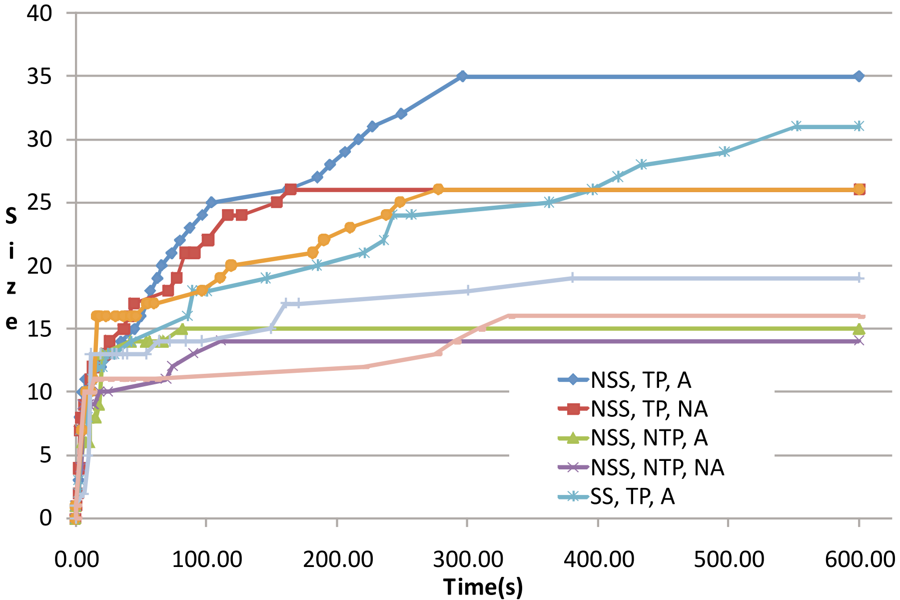

Fig. 8. Linked list: Optimizations (SS for SplitStructs, TP for two-phase solving, A for alignment and N for their negated counterparts)

9 Related Work

Recently, a broad range of test input generation tools have been developed based on symbolic execution [5,11,3,12,4,2]. We compare our work with them based on two dimensions:

Generating pointers as input. As we discussed in Section 3, most tools [5,11,3,12] focus on integration testing and do not support pointers as symbolic input for test methods (e.g. Sage [5] only focuses on files as input, and KLEE [12] on command-line arguments and files). For unit testing complex data structures this support is essential.

Some argue that a test that requires complex test inputs can be wrapped in another test that only takes (an array of) inputs with primitive types; the wrapping test first parses the complex data from the primitive data, and then calls the original test; this approach complicates the exploration of the code under test by requiring the exploration of the parser in addition, and it

also does not take into account possibly legal configurations of the complex data which are not generated by the parser.

An alternative solution is test sequence generation, the process of creating the data structure iteratively by starting with a constructor call followed with a list of methods with symbolic inputs. Unfortunately, test sequence generation is far from scalable in practice.

In this paper, we use a designated function to validate the data structure (also known as repOk methods). This function is similar to the concept of invariant in deductive software verification. For a valid test input, we assume that the data structure is valid, then we apply the function we want to test. Finally, we assert that the resulting data-structure is still valid.

Reasoning about pointers. There is much diversity in the way different tools handle pointers: e.g. Java Pathfinder [2] only supports object references.

Cute [4] does support pointers, they collect only (dis)equality constraints over these pointers. Although one might argue that using complicated operations on pointers is bad practice, it is used in rare but intricate cases. For example, we frequently encountered alignment-checks on pointers, where the lower bits of the pointer are inspected by the program. Our approach treats pointers as regular integers supporting all integer arithmetic operations.

SimC [13] is another tool to generate test inputs that support pointer reasoning. Unlike us, they model memory as one large array. SimC implicitly assumes that pointers do not overlap in incorrect ways.

Most tools perform concretization at some level to support pointer reasoning [3,5,4], i.e. they use the concrete value of a pointer as observed during concrete execution. Cute does not use the theory of arrays for representing memory but concretizes indices of array accesses. This leads to the inability to generate test inputs where $i == j$ in the following program: *a[i]=0; a[j]=1; if (a[i]==0) ERROR*. EXE concretizes pointers when there are double dereferences. Finally, Sage uses concretization to handle symbolic indexing into an array. We do not perform concretization to deal with pointers.

For program verification (of low level code), dealing with pointers is challenging as well. Most verification tools are incomplete with respect to pointers. For example, in VCC [14], pointers are treated as logical references instead of integers. Havoc [15] also treats pointers as integers. Havoc also has an encoding of the type system for SMT solvers [16], but our encoding is more precise. In particular, in Havoc, two different structs with fields of the same type can not be differentiated. Furthermore, they don't encode disjointness constraints at the byte level. Finally, separation logic [17] is a promising alternative way to reason about heap manipulating programs.

10 Conclusion

In this paper, we proposed a novel solution for generating test inputs for programs with pointers. We exploited the type information to encode disjointness

assumptions that characterize acceptable configurations of typed pointers in byte-addressable memory as constraints for the solver. As a result, the constraint solver only computes relevant heap shapes for the program under test.

We have implemented our approach in Pex, and evaluated it on red black trees and linked lists. From the two memory representations we created, the representation with a map per type was much faster than the representation where we only maintained a map per primitive type. Thanks to the two-phase solving optimization, the disjointness axioms have only minimal impact on the performance.

References

1. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
2. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with java pathfinder. In: ISSTA (2004)
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: CCS 2006 (2006)
4. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proc. of ESEC/FSE 2005, pp. 263–272. ACM Press, New York (2005)
5. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of NDSS 2008 (Network and Distributed Systems Security) (2008)
6. Tillmann, N., de Halleux, J.: Pex–white box test generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
7. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. SIGPLAN Notices 40(6), 213–223 (2005)
8. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Abstract synergy: A new algorithm for property checking (2006)
9. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: SOSP (2005)
10. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 2–23. Springer, Heidelberg (2005)
12. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008 (to appear)
13. Xu, Z., Zhang, J.: A test data generation tool for unit testing of c programs. QSIC 0, 107–116 (2006)
14. Schulte, W., Xia, S., Smans, J., Piessens, F.: A glimpse of a verifying c compiler – extended abstract (2007)
15. Chatterjee, S., Lahiri, S.K., Qadeer, S.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007)
16. Condit, J., Hackett, B., Lahiri, S., Qadeer, S.: Unifying type checking and property checking for low-level codes. In: POPL (to appear, 2009)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)