Result graphs for an abstract interpretation-based static analyzer

Pascal Cuoq, Raphaël Rieu-Helft

Result graphs for an abstract interpretation-based static analyzer

Pascal Cuoq[1] & Raphaël Rieu-Helft[2]

1: Trust-In-Soft
cuoq@trust-in-soft.com
2: École Normale Supérieure
raphael.rieu-helft@ens.fr

Abstract

TIS-Analyzer is a static analysis platform based on Frama-C. It integrates C analyzers in a plugin architecture and can be used to soundly detect undefined behaviors in C programs. The plugins communicate with each other to increase their precision. The Value analysis plugin uses dataflow analysis to produce a sound representation of the memory state at each control point of the program. Its abstract domain allows to represent disjunctions of non-relational value states. Further plugins then use this information to conduct derived analyses (operational inputs, dependencies...) However, the disjunctions alone are not sufficient: they do not allow the derived analyses to know which disjuncts of the representation of an abstract state can be reached from each disjunct of the state at the previous statement. We present a representation of the end result of the Value analysis as a graph that refines the program's control flow graph. In this "result graph", each separate non-relational abstract state occurring at a particular statement is represented as a distinct node. We argue that this representation is what derived analyses should work on by default. It is also a way to formalize the Value analysis in terms of abstract interpretation. Finally, result graphs are suited for human review and allow users to find the root cause of alarms raised by the analysis. When implemented naively, the propagation of the disjunctions of abstract states requires a quadratic number of potentially costly inclusion tests between abstract states. We present an efficient algorithm that takes advantage of the hash-consed representation of abstract states as Patricia trees to reduce the cost of testing the inclusion of a state in a set of states to amortized constant time. We also justify the correctness of this algorithm in a more general setting.

1. Context

1.1. C and undefined behaviors

Some erroneous actions in C, such as dividing an integer by 0 or dereferencing an uninitialized pointer are defined by the C standard as having an *undefined behavior*. When present in a C program, these actions cause the behavior of the entire program to be unspecified as far as the standard is concerned. The C FAQ[1] defines the notion of undefined behavior like this:

This work was partially supported by ANR-14-CE28-0014 AnaStaSec.

[1] Steve Summit. C FAQ on undefined behaviors. http://c-faq.com/ansi/undef.html

> Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.

The main advantage of such a loose specification for incorrect actions is that C compilers may "trust" the programmer and assume that their input does not have an undefined behavior. Not only do they not have to check for them, but this assumption also gives them more information to implement more powerful optimizations. In the event that their input does yield an undefined behavior, then whatever binary they output is correct by default.

For example, signed integer overflow is undefined in C; compilers therefore do not have to worry about the case where a variable in a loop overflows, which leads to significant speed-ups in performance-critical loops.

However, an important drawback of undefined behaviors is that programmers cannot be trusted to reliably avoid them. This leads to programs silently exhibiting an incorrect behavior. This problem is compounded by the compilers' aggressive optimizations: debugging optimized code can prove challenging. Consider this code extracted from an erroneous bug report for the compiler gcc [1]:

```
1 void foo(int a) {
2   assert(a+100 > a);
3   printf("%d␣%d\n",a+100,a);
4 }
5
6 void main() {
7   foo(0x7fffffff);
8 }
```

The programmer's expectation is that the program can never output two decreasingly ordered numbers (otherwise the assertion would fail). On the particular input given by `main` to `foo`, there is an integer overflow (in a 32-bit architecture). A natural outcome is that `a + 100` wraps around to a large negative integer and that the assertion fails. However, when compiled with the usual command `gcc -O2` (for gcc version 4.8.4), the output is:

```
> ./a.out
-2147483549 2147483647
```

The explanation is that signed integer overflow is undefined in C. Therefore, when considering the assertion at line 2, the compiler may assume that `a + 100` does not exceed the maximum integer. Therefore, the assertion is always verified, and it is optimized away! The assembly output is the same as if the assertion was not in the code in the first place. This causes the program to silently fail even though the programmer included an assertion specifically to avoid this. Instead, the programmer should either implement a different check or tell the compiler to use a wrap-around semantics for integer overflow with the flag `-fwrapv`, at a potential performance cost.

This sort of compiler behavior pressures programmers to be particularly careful to avoid undefined behaviors. However, they can be hard to spot, and the C standard lists over a hundred different kinds of undefined behaviors.

1.2. TIS-Analyzer overview

Frama-C and its derivative TIS-Analyzer are static analysis platforms for C programs, and are developed in OCaml. They integrate several C static analyzers in a plugin architecture: they share information through interfaces with a common kernel that conducts the analysis. Each plugin can access the results that previous plugins computed to increase precision and avoid redundant

computations. This architecture also allows advanced users to develop their own plugins and integrate them into the platform, accessing the results of the other plugins for their own analyses with relatively little effort. This makes it possible for the platform to detect most kinds of undefined behavior as well as to operate program transformations such as slicing.

In addition to the kernel interfaces, Frama-C and TIS-Analyzer use the ANSI/ISO C Specification Language (ACSL) to express function specifications and proof obligations [2]. Users may annotate the source code with ACSL annotations and function contracts, and plugins may emit alarms in the form of proof obligations that are preconditions to the source program's correctness. Conversely, the WP plugin uses Hoare-style weakest precondition calculus to prove ACSL properties by leveraging the results of other plugins as well as external theorem provers such as Alt-Ergo [4] or Why3 [5].

1.3. The Value analysis

Overview The Value analysis plug-in is central to TIS-Analyzer. It is an abstract interpreter that uses dataflow analysis to compute the values of each variable at each program point, and detects related undefined behaviors along the way. When the analysis cannot prove the absence of an undefined behavior, an alarm is emitted as an ACSL proof obligation. The analysis then continues under the assumption that the proof obligation is satisfied (that is, that no undefined behavior occurred). It would be desirable to predict what happens after the undefined behavior, but as the example from 1.1 shows, it is folly to attempt to predict what happens in an execution that contains undefined behavior.

Semantic unrolling The Value plugin performs a forward dataflow analysis, which means that it computes a fixpoint on the control flow graph of the source program starting at the initial statement. The state of the program's memory along an execution trace is represented as an element of a non-relational abstract domain. Possible values for integers are represented as sets when the number of possible values is small. When there are many possible values, they are abstracted in a compact representation as intervals with congruence information (e.g. "all the values between -6 and 34 that are congruent to 2 modulo 4"). Floating-point values are always abstracted as intervals. Both these abstractions obviously pollute the feasible values with unfeasible ones. This should not distract the reader from the loss of information intrinsic to the choice of a non-relational representation where the values of each variable, array element and struct member is represented without regard to its correlation to other variables, array elements and struct members.

Using the simple dataflow analysis algorithm [10] with the above non-relational representation generates spurious alarms for many programs. Here is an example:

```
1  int x,y,s;
2  x = rand() % 18 - 9
3  // x = [-9,8]
4  if (x >= 0)
5    s = 1;
6  else {
7    s = -1;
8  }
9  //s={-1; 1}, x=[-9,8]
10 y = x*s;
11 //s={-1; 1}, x=[-9,8], y=[-9,9]
12 assert (y>=0); //alarm!
```

In this example, the assertion is always verified during real execution, but this cannot be proved with a non-relational abstract domain. After the conditional, the memory states at the end of lines 4 and 6 are joined and the possible values for `s` are -1 and 1, while `x` can be anything between -9 and 8.

While there is in fact a relation between these values (they have the same sign), the domain is unable to express this.

To mitigate this loss of information, the Value analysis offers the possibility to represent the values of variables `x`, `y`, `s` as a disjunction of several non-relational memory states originating from different traces. For if-then-else constructs, this means that the state coming out of the "then" branch is propagated separately from the state from the "else" branch. In the example, the two states $(x \in [0,8], s = 1, y \in [0,8])$ and $(x \in [-9,-1], s = -1, y \in [1,9])$ reach and pass the assertion independently, and the analysis validates it. More precisely, the disjunctions allow the analysis to be more precise on the value of y and to discover that $y \in [0,9]$ at the end of the program. Loops can be unrolled through the same mechanism: the values in memory after two iterations are stored separately from the values after one iteration, etc. limiting the confusion between how the values are correlated as the loop progresses in a real execution.

Obtaining a fixpoint In order for a fixpoint to be guaranteed to be produced in finite time, it is necessary to limit the number of states that can be maintained in the disjunction associated to a statement. This is done with the option `-slevel`, which sets the maximum cardinal of the disjunctions. When a new state is added to a disjunction and the `slevel` is reached, the disjuncts are all merged and the algorithm reverts to the simple forward dataflow analysis behavior. This is effectively a widening.

Setting the `slevel` to 2 or more allows to analyze the previous example precisely: before the assignment at line 9, the state is a disjunction of the states coming from the `then` and `else` branch. The assignment is applied to both states independently, and both resulting states satisfy the assertion.

For efficiency, techniques taking inspiration from explicit-state model-checking are applied. This is similar to "Configurable Software Verification" [3] but was developed independently. The present article documents implementation aspects that are not covered in Beyer, Henzinger and Théoduloz's article.

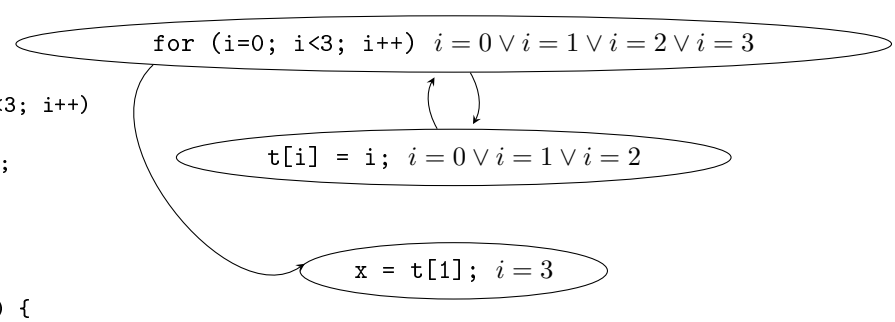
When an additional non-relational state is propagated to a statement, this state is first tested for inclusion against all the non-relational states that have already been associated to this statement. If the new memory state is included in a state previously associated to the statement, then it does not add any new information and can be discarded. This saves time in the sense that the memory state does not need to be propagated. This also gains precision as it avoids inserting a non-informative disjunct in the disjunction, and thus delays the join. An example where time and precision are gained is `for (char i=0, j=1; i!=j; i++,j++);`, where the analyzer discovers that the program state after 256 iterations, in which `i` is 0 and `j` is 1, is included in the initial state, and thus concludes that the loop is infinite without reaching the slevel limit at which it would have merged all states and lost the opportunity to conclude that the loop condition is always true.

In a naïve implementation of the propagation algorithm, a quadratic number of inclusion tests would be necessary. Section 4 shows how this quadratic behavior of the analyzer is avoided in practice, building on top of previous efforts for efficient representation of memory states with hash-consing[7].

1.4. Derived analyses: a motivating example

The results of the value analysis may then be used by other plugins. The operational inputs (or *inout*) plugin is one of them. It is a derived analysis of Value, and computes the operational inputs and sure outputs of each function call in a C program. The operational inputs of a function are defined as the memory zones that may be read by the function without having been previously overwritten, and the sure outputs are the memory zones that are written to for sure. The inout plugin runs a dataflow analysis on the control flow graph, with knowledge of the results of Value. The following example shows why expressing the results of Value in terms of a disjunction of abstract memory states per program statement is not sufficiently precise.

```
1  char t[3];
2  void f() {
3    int i,x;
4    for (i=0; i<3; i++)
5      {
6        t[i] = i;
7      }
8    x = t[1];
9    }
10
11 int main(void) {
12   f();
13 }
```



Control flow graph of `f`, decorated with Value results

Figure 1: Initializing an array with a `for` loop

Consider the function in Figure 1. It initializes an array with a `for` loop and then reads from a cell. The most precise result for the inout analysis is that the whole array is in the sure outputs of `f`, and that the operational inputs are empty: the only location we read from has been written to beforehand in the function.

However, if the inout analysis only had access to the information in the decorated control flow graph from the figure, the results would be less precise. The explanation is that the control flow graph of the program is not expressive enough to enforce that the program goes through the full three iterations of the `for` loop. The inout analysis would therefore conclude that the program might read `t[1]` without having assigned to it, which would make it an operational input.

What is missing here is that the only way to reach the assignment is from the $i = 3$ disjunct of the `for` loop, which is itself only reachable after having gone through three iterations, during one of which `t[1]` was set to the value of an expression, `i`, itself without dependencies. The transitions between states are available to Value. It is important that these transitions be made available to derived analyses, as the example shows. We argue that the graph of states, with the transitions between them, should be considered the result of the analysis, and should be the form that derived analyses work on by default.

2. Result graphs

2.1. Principle

The nodes of the value analysis' result graph are (statement, abstract state) pairs. The disjunction of possible abstract memory states at a statement is represented by as many nodes as the statement has disjuncts. Edges of the result graph track which disjunct at a statement can reach which disjuncts at the successor statements.

When a statement's transfer function is applied to a state and results in new states at the successor statements, the value analyzer creates new nodes for the resulting states at these statements. Edges are inserted between the original node and each of the new nodes.

For example, on the program from Figure 1, the result graph at the end of the value analysis would be the one in Figure 2. The graph shows that the only way to reach the assignment at the end of the program is after three iterations of the `for` loop. Running the inout analysis on this graph instead of the control flow graph of the program fixes the imprecision we warned about in Section 1.4.

The graph representation is also a good fit to represent the other features described in Section 1.3.

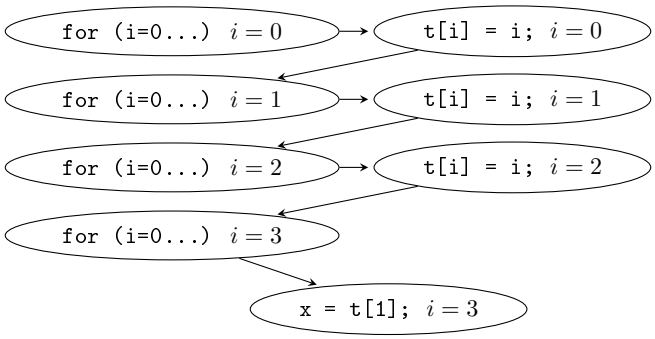


Figure 2: Result graph for `f` in Figure 1

When a new state is discovered, if a greater state already exists, we can simply add an edge to the corresponding node instead of creating a new node. Similarly, when the `slevel` is reached and a disjunction is merged, we can create a node for the joined state and add it as a successor of all the states that had a successor in the disjunction.

The approach is similar to trace partitioning [12] in the sense that it increases precision by delaying joins and tracking where each disjunct of the abstract state comes from in terms of control flow. One key difference is that in trace partitioning, states produced by different execution paths are considered different by virtue of coming from different paths, even if they assign the same values to all the program's variables. However, as outlined in Section 1.3, result graphs merge states when possible at the same program point: the second state to reach the junction point is considered to be subsumed by the first one. Even a state that is not equal to, but merely included in a previously encountered state is never propagated. This feature is closer to model checking [3].

2.2. Formalization

We define a framework to enhance an existing abstract interpreter and adapt it to compute result graphs, with some simple assumptions on the underlying abstract domain. This formalizes the approach we used on the Value analysis and proves its soundness. This also provides a formal definition of the semantic unrolling strategy in terms of abstract interpretation, which had never been done before to the best of our knowledge.

Notations We introduce a few notations. Let:

- $\mathbb{X}$ a set of variables
- $\mathbb{V}$ the set of values
- $\mathbb{M} = \mathbb{X} \to \mathbb{V}$ the set of memory states
- $\mathbb{L}$ the set of control states (program counter)
- $\mathbb{S} = \mathbb{L} \times \mathbb{M}$

A program is represented by a tuple $(\mathbb{X}, \mathbb{L}, \mathbb{S}^i, \to)$ where $\mathbb{S}^i$ is the set of initial states, and the transition relation $(\to) \subseteq \mathbb{S} \times \mathbb{S}$ describes how the execution goes from one state to another.

We assume here that the programs have no function calls in order to simplify the notations. When they do, we have to add a notion of callstacks but there is no conceptual difficulty. We also do some simple rewriting on the analyzed programs so that they have exactly one entry point $l_\vdash$ and one exit point $l_\dashv$.

Concrete semantics We define a trace semantics for programs:

Definition 1. *Trace semantics*

A trace is a finite sequence of states in $\mathbb{S}$*. We write* $\mathbb{S}^*$ *for the set of traces. For a program* $P = (\mathbb{X}, \mathbb{L}, \mathbb{S}^i, \rightarrow)$*, we define its trace semantics* $[\![P]\!]$ *as the set of traces that start at a state in* $\mathbb{S}^i$ *and are valid under the relation* $\rightarrow$*:* $[\![P]\!] = \left\{ \langle s_0, \ldots, s_n \rangle \;\middle|\; \exists m_0 \in \mathbb{M}, s_0 \in \mathbb{S}^i \wedge (l_\vdash, m_0) = s_0 \rightarrow \ldots \rightarrow s_n \right\}$.

Note that unfinished executions (that do not end at $l_\dashv$) are accepted by the former definition. However, infinite traces are not.

Abstract domain We assume that an abstract domain for memory states $(D^\sharp, \sqsubseteq)$ is defined, along with a concretization function $\gamma_{\mathbb{M}} : D^\sharp \rightarrow \mathcal{P}(\mathbb{M})$. We write $S^\sharp$ for $\mathbb{L} \times D^\sharp$.

We also require that $D^\sharp$ be equipped with an abstract join operator $\sqcup$ that soundly approximates the union in $\mathcal{P}(\mathbb{M})$, and sound and monotone transfer functions for all the instructions in the language. We also require for $D^\sharp$ to have a least element $\bot$ such that $\gamma_{\mathbb{M}}(\bot) = \emptyset$.

We define the elements of a new abstract domain $\mathbb{D}^\sharp$ as tuples of the form $(\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \rightarrow^\sharp, \Phi^\sharp)$ as elements, where $S^{\sharp^i} \subseteq \{ l_\vdash \} \times D^\sharp$ is a set of abstract initial states, $(\rightarrow^\sharp) \subseteq S^\sharp \times S^\sharp$ is the transition relation, and $\Phi^\sharp : \mathbb{L} \rightarrow \mathcal{P}(D^\sharp)$ is such that if $(l, \sigma^\sharp)$ appears in $(\rightarrow^\sharp)$, $\sigma^\sharp \in \Phi^\sharp(l)$. Φ represents the set of possible stores at each control state. The order $\sqsubseteq$ on $\mathbb{D}^\sharp$ is pointwise inclusion.

An element of $\mathbb{D}^\sharp$ can be concretized into a set of traces by applying $\gamma_{\mathbb{M}}$ pointwise on $\Phi^\sharp$ and $\rightarrow^\sharp$ and then looking at paths in the resulting transition relation:

$$\gamma^\sharp_{\mathbb{D}}(\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \rightarrow^\sharp, \Phi^\sharp) = \left\{ \langle s_0, \ldots, s_n \rangle \;\middle|\; \exists m_0 \in \mathbb{M}, s_0 \in \mathbb{S}^i \wedge (l_\vdash, m_0) = s_0 \rightarrow \ldots \rightarrow s_n \right\}$$

$$\text{with: } \begin{cases} \mathbb{S}^i & = \bigcup_{d \in S^{\sharp^i}} \gamma_{\mathbb{M}}(d) \\ \rightarrow & = \bigcup_{d_1 \rightarrow^\sharp d_2} \{ (s_1, s_2) \mid s_1 \in \gamma_{\mathbb{M}}(d_1) \wedge s_2 \in \gamma_{\mathbb{M}}(d_2) \} \end{cases}$$

Elements of $\mathbb{D}^\sharp$ should be seen as refined versions of the control flow graph. The semantics of statements are transfer functions in $\mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$ that enrich the graph by adding new edges.

Semantic unrolling We maintain a unicity invariant: if $\sigma^\sharp \in S^\sharp$, no new state included in $\sigma^\sharp$ can be added to $S^\sharp$. This is the strategy discussed at the end of Section 1.3.

To enforce this, we need to check for inclusion each time a new abstract store is computed. We define an operator to denote this inclusion check:

$$\texttt{greater}(d, S) = \begin{cases} d & \text{if } \forall d' \in S, d \not\sqsubseteq d' \\ d' & \text{if } d' \in S \wedge d \sqsubseteq d' \end{cases}$$

This means that instead of adding an edge to a new state d to S, we can add an edge to $\texttt{greater}(d, S)$. This is sound because for all d, S, $d \sqsubseteq \texttt{greater}(d, S)$. The operator is non-deterministic: the choice of the greater state to return is arbitrary if several exist. However, this choice has no influence on the rest of the analysis: if it has to be made, there is no new information to propagate anyway.

We can now define an operator that merges a new edge into an element of $\mathbb{D}^\sharp$ while maintaining the unicity invariant:

$$\texttt{add} : (S^\sharp \times S^\sharp) \times \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$$
$$(((l_1, d_1), (l_2, d_2)), (\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \rightarrow^\sharp, \Phi^\sharp)) \mapsto (\mathbb{X}, \mathbb{L}, S^{\sharp^i}, (\rightarrow^\sharp \cup ((l_1, d_1), (l_2, d))), \Phi'^\sharp)$$

where

$$\begin{cases} d & = \texttt{greater}(d_2, \Phi^\sharp(l_2)) \\ \Phi'^\sharp & = \begin{cases} l_2 & \mapsto \{\, d \,\} \cup \Phi^\sharp(l_2) \\ l \neq l_2 & \mapsto \Phi^\sharp(l) \end{cases} \end{cases}$$

By abuse of notation, we extend `add` to take a set of new edges as its first parameter and add them to a graph in an arbitrary order.

Transfer functions Let us now define transfer functions for statements. We require, for each control point $l \in \mathbb{L}$, a function $[\![l]\!]^\sharp_D : S^\sharp \to \mathcal{P}(S^\sharp)$. This transfer function is then extended into

$$[\![l]\!]^\sharp_{\mathbb{D}} : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$$
$$g = (\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \to^\sharp, \Phi^\sharp) \mapsto \texttt{add}(\bigcup_{d \in \Phi^\sharp(l)} [\![l]\!]^\sharp_D(d), g)$$

$[\![l]\!]^\sharp_{\mathbb{D}}$ is a sound approximation of the concrete semantics $[\![l]\!]$ of the statement s at the control point l provided $[\![l]\!]^\sharp_D$ is.

Slevel limitation Lastly, we need to guarantee that the analysis terminates in a finite number of iterations. We assume that $D^\sharp$ is equipped with a widening operator $\nabla_{\mathbb{M}}$. We set a maximum number N_l of abstract states in each $\Phi(l), l \in \mathbb{L}$. Upon adding a new abstract store at a control point l where N_l abstract stores or more are already present, we merge them: in the new state, there is only one abstract store at control point l, which is the join of the previous ones. This corresponds to the maximum number of states in the disjunctions of Section 1.3. From that point on, new abstract stores at this control point are merged into that one with $\nabla_{\mathbb{M}}$ instead of being added to the disjunction. As $\nabla_{\mathbb{M}}$ guarantees that we cannot encounter an infinitely increasing sequence of states, this guarantees that the fixed point is reached after a finite number of iterations. This is a widening operator on $\mathbb{D}^\sharp$.

More precisely, we define an operator $\texttt{add}_\nabla : (S^\sharp \times S^\sharp) \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ that adds an edge to an abstract state while maintaining this invariant:

$$\texttt{add}_\nabla(((l_1, d_1), (l_2, d_2)), (\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \to^\sharp, \Phi^\sharp))) =$$
$$\begin{cases} \texttt{add}(((l_1, d_1), (l_2, d_2)), (\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \to^\sharp, \Phi^\sharp))) \text{ if } |\Phi^\sharp(l_2)| < N_{l_2} \\ \texttt{add}(\{\, ((l_1, d_1), (l_2, d_0)) \,\} \cup \left\{\, ((l, d), (l_2, d_0)) \;\middle|\; \exists d', (l, d) \to^\sharp (l_2, d') \,\right\}, (\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \to^\sharp, \Phi^\sharp)) \\ \quad \text{otherwise, with } d_0 = \bigsqcup \left\{\, d \;\middle|\; d \in \Phi^\sharp(l_2) \,\right\} \nabla_{\mathbb{M}}\, d_2 \end{cases}$$

We then define the widening operator for $\mathbb{D}^\sharp$ as a fold of $\texttt{add}_\nabla$ in an arbitrary order on all the edges of its second argument:

$$\nabla(x^\sharp, (\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \to^\sharp_2, \Phi^\sharp_2)) = \texttt{add}_\nabla\left(\left\{ \bigcup_{l_1, l_2 \in \mathbb{L}} \left\{\, (d_1, d_2) \;\middle|\; d_1 \to^\sharp_2 d_2, d_1 \in \Phi^\sharp_2(l_1), d_2 \in \Phi^\sharp_2(l_2) \,\right\} \right\}, x^\sharp\right)$$

We only define ∇ when its two arguments have the same first three fields (it is always the case for a given program).

Theorem 1. *∇ is a widening operator.*

- $\forall x^\sharp, y^\sharp, x^\sharp \sqsubseteq x^\sharp \nabla y^\sharp$ *and* $y^\sharp \sqsubseteq x^\sharp \nabla y^\sharp$.

- *For any sequence $(x_n^\sharp)_{n\in\mathbb{N}}$, the following sequence $(y_n^\sharp)_{n\in\mathbb{N}}$ is not strictly increasing:*

$$\begin{cases} & y_0^\sharp = x_0^\sharp \\ \forall n \in \mathbb{N}, & y_{n+1}^\sharp = y_n^\sharp \nabla x_{n+1}^\sharp \end{cases}$$

Proof.

- Let $x^\sharp, y^\sharp \in \mathbb{D}^\sharp$. Clearly $x^\sharp \sqsubseteq x^\sharp \nabla y^\sharp$ as `add` is monotone, and $y^\sharp \sqsubseteq x^\sharp \nabla y^\sharp$ because each edge in $y^\sharp$ was added to $x^\sharp \nabla y^\sharp$ (and the contents of $\Phi^\sharp$ was adjusted accordingly by `add`).

- Let $(x_n^\sharp)_{n\in\mathbb{N}}$ a sequence in D and let us define $(y_n)_{n\in\mathbb{N}}$ as previously and $(\Phi_n^\sharp)_{n\in\mathbb{N}}$ the sequence of the $\Phi^\sharp$ functions of the (y_n). If (y_n) is strictly increasing, then for some l, $(\Phi_n^\sharp(l))_{n\in\mathbb{N}}$ must be strictly increasing. After a certain rank, $|\Phi^\sharp(l)| > N_l$. After this, $(\Phi^\sharp(l))_n$ is a subsequence of sequence of iterates of $\nabla_{\mathbb{M}}$, so it cannot be strictly increasing, which is a contradiction.

□

The strategy that consists in merging all the abstract stores at a control point when the `slevel` is reached is sound, but is not always the most precise. For example, one could imagine privileging a particular variable and trying to merge together the states that associate the same abstract value to it, so as to not lose precision for this particular variable. The exploration of other possible merging strategies is left for future work.

Abstract semantics We now define the abstract semantics of a program $(\mathbb{X}, \mathbb{L}, \mathbb{S}^i, \rightarrow)$ as the least solution $x^\sharp = (\mathbb{X}, \mathbb{L}, S^{\sharp^i}, \rightarrow^\sharp, \Phi^\sharp)$ of the following equation system defined from the transfer functions:

$$\begin{cases} \mathbb{S}^i \subseteq \bigcup\limits_{d \in S^{\sharp^i}} \gamma_{\mathbb{M}}(d) & (1) \\ S^{\sharp^i} \subseteq \Phi^\sharp(l_\vdash) & (2) \\ \forall l \in \mathbb{L}, x^\sharp = x^\sharp \nabla [\![l]\!]_{\mathbb{D}}^\sharp(x^\sharp) & (3) \end{cases}$$

Theorem 2. *Soundness.*
If $x^\sharp$ is a solution of the equation system above for an input program P, then $[\![P]\!] \subseteq \gamma_{\mathbb{D}}^\sharp(x^\sharp)$.

Proof. By induction on the length of the trace in $[\![P]\!]$. Traces in $[\![P]\!]$ of length 1 are in $\gamma_{\mathbb{D}}^\sharp(x^\sharp)$ because of (1).
Let us remark that $[\![P]\!]$ is prefix-closed. Let $\langle s_0, s_1, \ldots, s_{n-1}, s_n\rangle \in [\![P]\!]$. Then $\langle s_0, \ldots, s_{n-1}\rangle \in [\![P]\!]$, and $\langle s_0, \ldots, s_{n-1}\rangle \in \gamma_{\mathbb{D}}^\sharp(x^\sharp)$ by induction hypothesis. Let $(l, m) = s_n$. The soundness of the transfer function $[\![l]\!]_{\mathbb{D}}^\sharp$ and (3) mean that $s_n \in \Phi^\sharp(l)$ and $\langle s_0, \ldots, s_n\rangle \in \gamma_{\mathbb{D}}^\sharp(x^\sharp)$ (using the monotonicity of ∇), which concludes the proof. □

In our implementation, the solution is computed using a worklist algorithm [9]. We can guarantee that the execution terminates in a finite number of steps thanks to the use of widening, by adapting the proof of Theorem 1.

3. Applications

3.1. Precision gains in derived analyses

We adapted some TIS-Analyzer plugins to take advantage of the result graph domain instead of working on disjunctions of abstract memory states. This led to precision gains in numerous ways. First and foremost, control flow imprecisions due to the derived analyses not being aware of the edges of the result graph disappeared (see section 1.4). Another precision gain is that dead paths are now properly ignored when computing the state at the end of a function. Indeed, when Value finds a potential undefined behavior, it emits an alarm and a proof obligation and then reduces the abstract state by assuming that the alarm is a false positive: in the case where it's not, it has already emitted an alarm, so this is always correct.

Consider the program in Figure 3. If `cond` is true, it attempts to read the cell 12 of an array of size 10. This is an out-of-bounds read, which is an undefined behavior. Therefore, the state in the "then" branch is reduced to bottom (the program must yield an undefined behavior if the "then" branch is taken, so it is not worth propagating further). Since the state is bottom, it is not propagated. The state at line 6 is therefore the state from the "else" branch. Therefore, the abstract state at the end of the program contains `cond` $= 0$. This is visible in the result graph (Figure 4): since the state at the "then" branch is not propagated (no outgoing edges), the only way to a `return` is through the "else" branch (which has `cond` $= 0$).

Therefore, the dependency analysis (see section 3.2) is now aware that `x` does not depend on `cond`.

```
1 int x,T[10];
2
3 void main(int cond) {
4   x = 42;
5   if (cond) x = T[12]; //UB
6   return; //cond must be 0
7 }
```

Figure 3: Dead path due to an invalid read

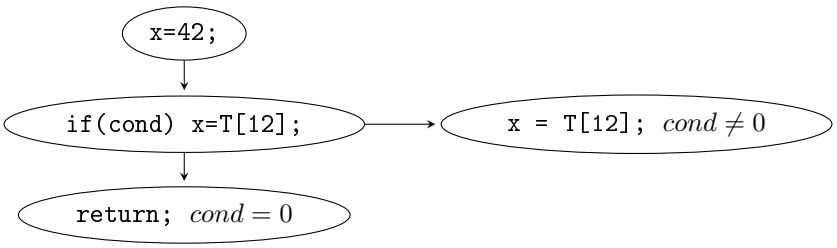


Figure 4: Dead path in the result graph

3.2. Whole-program dependencies

A previous version of the dependency analysis is implemented by the "From" TIS-Analyzer plugin. For each variable in a function, the dependency analysis computes from which variables it depends at the end of the function. This is computed by a forward dataflow analysis on the program's control flow graph using simple syntaxic rules. It keeps track both of data dependencies (which values were used to compute the value of the variable) and control flow dependencies (which variables the control flow choices made to reach that program point depend on). This dependency analysis only works within

the scope of a function: it computes the dependencies of each variable in the function with respect to the variables that were initialized at the beginning of the function call (parameters, globals...)

The result graph domain allowed us to develop a whole-program dependency analysis. It operates similarly to the existing From plugin, but the dataflow analysis is computed on the result graph instead of the control flow graph, and on the whole program rather than on each function separately. The dependencies are thus expressed in terms of the inputs of the program, which has a number of useful applications. A whole-program analysis would have theoretically been possible without using result graphs, but on the scale of a whole program, control flow imprecisions such as in Section 1.4 were impactful enough to make it useless in practice. We were able to use our analysis on two real-life libraries: libksba and mbed TLS.

3.3. Investigating a bug in libksba

Libksba is a X.509 certificate parser. The functions we investigated take a certificate as input, parse it, and check the validity of its signature. The input is typically a binary file of around 1 kB in size. The dependency analysis of section 3.2 allows us, for any program point, to express the dependencies of any variable in terms of specific bytes of the input certificate. As the analysis is sound, modifying a byte of the input that is not in the dependencies will not change the value of the variable.

We used this to investigate a known bug on an input obtained by fuzzing (see Section 3.4). This input is an invalid certificate of around 1 kB in size, and libksba did reject it, but took over a minute and a half to do it [6] when it should have been almost instantaneous. This is a security vulnerability: an attacker could perform a denial-of-service attack on a server using libksba by sending this certificate repeatedly.

Profiling revealed that most of the time was spent in a call to the C builtin `calloc`, which allocates a block in memory and initializes it to 0, which is time-consuming when the block is large. The size of the block was determined during the parsing of the certificate, but what exactly controlled it was unclear in the code. However, the data dependencies of the length variable happened to be exactly a block of six bytes in the input certificate. Furthermore, the length of the block to allocate was exactly the value written in the six-byte block seen as a 48-digit number.

Testing confirmed that writing an arbitrary length in these six bytes and not changing the rest of the certificate made libksba allocate a block with that exact length. This also means that the attacker could optimize the denial of service by requiring a block of the largest possible size that the server has available (asking for more would make the call to `calloc` fail immediately).

This was a good example of precise information obtained with our new dependency analysis. While someone with good knowledge of X.509 certificates may have been able to know or find where the block size was written in the certificate without the help of static analysis, we were able to do it in a matter of minutes and with no prior knowledge of X.509 certificates.

3.4. Fuzzing the mbed TLS library

Fuzzing is a testing technique that involves providing many randomized inputs to a program to find bugs in it. The fuzzer we used[2] is mutation-based, which means it randomly modifies bytes of a starting input. It uses genetic algorithms to learn which parts of the input are most interesting to modify to increase code coverage, but the process is still largely random.

We used our dependency analysis in conjunction with the fuzzer to investigate the cryptographic library mbed TLS more efficiently. The analysis keeps track of the control flow dependencies, which are the dependencies of all the conditionals that are still open at the current statement (in the sense

[2] Michał Zalewski. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl/`

that there is a path from the conditional to the end of the program that does not go through the current statement). Again, these dependencies are expressed in terms of the inputs of the program.

The soundness of the dependency analysis implies that for a given statement that is reached during the execution on an input, if we fuzz only zones of the input that are not in its control flow dependencies, we can guarantee that the statement is still covered by the fuzzed input. This is extremely useful in the context of fuzzing: when we are interested in investigating a specific function of the library, we can ensure that all inputs generated by the fuzzer will cover it by forbidding it to change the bytes of the input that are in the control flow dependencies at the entry point of the function. In our case, the fuzzer only very rarely generated an input that reached a particular function of mbed TLS. Locking in the control flow dependencies, which comprised around 20% of the input, allowed us to generate only test cases that reached that function and gain coverage much more rapidly.

4. Detecting inclusion of a state in a set of states

Implementing the `slevel` option requires efficient detection that a state newly propagated to a statement is included in any of the states that have already been propagated to that statement earlier. This step is necessary in order to detect that a fixpoint has been reached during the analysis of a loop that does not obviously terminate, and saves time that would have been wasted propagating the abstract states originating from both the then-branch and the else-branch of a conditional, if it happens that one is included in the other.

4.1. Principle

Analogously with the notion of finite elements, we call *singleton* elements of a lattice $(D, \sqsubseteq)$ with minimal element $\bot$ the non-bottom elements of D such that no non-bottom element of D is strictly smaller than them: x is a singleton if and only if $x \neq \bot$ and for all $y \in D, y \sqsubset x \implies y = \bot$.

In other words, the singleton elements of D are the minimal elements of the partially ordered set $(D \setminus \bot, \sqsubseteq)$. The intuition is that for a set X, the singleton elements of the powerset lattice $(P(X), \subseteq)$ are the sets that are singletons.

Let $(D, \sqsubseteq)$ be the lattice of abstract memory states with a minimal element $\bot$, and $(D^\sharp, \sqsubseteq^\sharp)$ another lattice with a minimal element $\bot^\sharp$. Let $\pi : D \to D^\sharp$ be a monotone function. We also require that for all $x \in D$, $x \neq \bot \implies \pi(x) \neq \bot^\sharp$.

Let us now assume that $x \in D$ is such that $\pi(x)$ is a singleton of $(D^\sharp, \sqsubseteq^\sharp)$. Then for all non-bottom $y \in D$, $y \sqsubseteq x \implies \pi(y) = \pi(x)$.

Efficiently testing whether a new state is included in one of the previous states during the value analysis relies on the contraposition of this implication: if $\pi(s_1) \neq \pi(s_2)$, there is no need to test whether $s_1 \sqsubseteq s_2$ because it cannot be true. In the value analysis, memory states are represented as Patricia trees and the function π consists in selecting the subtree located at a well-chosen path P. Among the previously seen states s_2, those such that $\pi(s_2)$ is singleton have been indexed in a hash-table, using each respective $\pi(s_2)$ as key. Out of these previous states, only the states such that $\pi(s_1) = \pi(s_2)$ need to be tested to see if they include s_1.

4.2. Implementation

The value analysis' states are represented as Patricia trees [11]. A subtree of the representation of an abstract state is itself the representation of an abstract state (the restriction of the former to a subset of the variables it defines). We call *singleton subtrees* the subtrees that characterize the variables that they define so precisely that their concrete values are known precisely. They are the singleton

elements (in the sense of Section 4.1) of the lattice of Value states.

Let us provide a description of the data structures involved using Ocaml syntax.

```
type variable
type value
type path
type tag = int

type state =
  | Empty
  | Leaf of variable * value * tag * bool
  | Branch of int * int * state * state * tag * bool

type subtree = state

type superposition =
    { table : (subtree, state) Hashtbl.t;
      path : path;
      others : state list }
```

The `state` type represents the Patricia trees used to describe the abstract state. The `tag` of each non-empty subtree is a unique hash-consing identifier[8]: two subtrees are equal iff the tags of their roots are equal. This means that they can be used as hashes and as a total order on the nodes. The two integers in the `Branch` constructor are part of the usual metadata of Patricia trees. Hashing and comparisons on trees are constant-time as a result. The boolean flag, which indicates whether a subtree is singleton, is computed from the children's own flags each time a new node is created (only a join of singleton subtrees is a singleton subtree). This helps to make it quicker to pick a suitable path (see next paragraph) and allows other optimizations[3].

States are indexed in the `superposition` structure. The `path` field stores a well-chosen path P. The `table` stores in a hashtable all the states with a singleton subtree at P, the keys being their subtrees at P. Finally, the `others` list stores all the states that do not have a singleton as subtree at P.

Example trees are given in Figure 5 (the first two integer fields of `Branch` nodes are omitted).

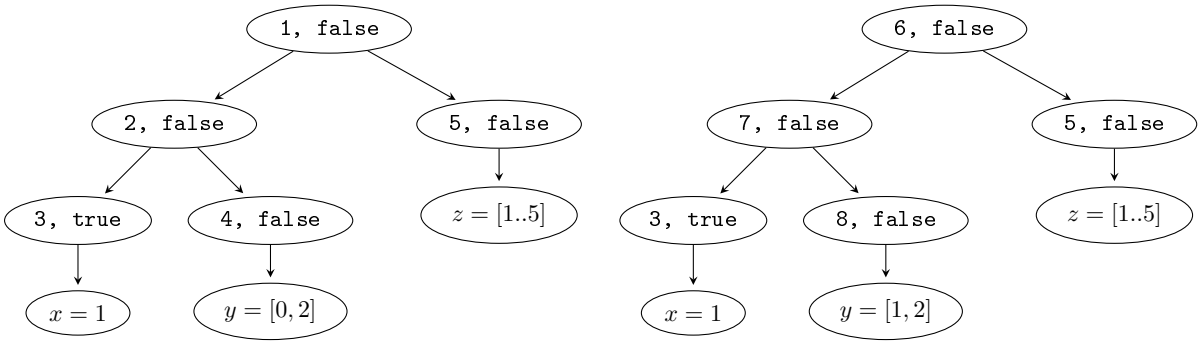


Figure 5: An example. The second tree is included in but not equal to the first one.

Assume for now that all the abstract states being handled have singleton subtrees at a certain path P. Under these ideal circumstances, each state's subtree at P can be used as a key according to which the entire state can be classified. A new state, candidate for being included in one of the previously seen states, only needs to be tested for inclusion against the states that have the exact same subtree at P.

[3] In the implementation of the inclusion test between states `s` and `t`, if `t` is singleton, then the answer can be computed immediately as `s == t` without having to recurse.

Ideally, all the states that eventually need to be handled at a given statement have differing singleton subtrees at P: this makes the inclusion test of a new state N into the set of previous states amortized constant-time. Specifically, the steps that occur are these:

- The bucket number for N is computed. This is very cheap because subtrees are numbered at creation and the number of a subtree is used as its hash.
- For each binding present in the bucket, an equality test is made between the subtree N has at P and previous subtrees from the bucket. This equality test again is very cheap thanks to hash-consing.
- For each state in the set of states associated to the subtree identical to that of N, a full inclusion test is made.

We have hitherto ignored the case where the new state N does not have a singleton subtree at P. In this case it is certain that N is not included in any of the states that have a singleton subtree at P. The issue is that of states that will arrive after N: a state M arriving after N may have singleton subtree at P. M will thus need to be tested for inclusion in N, even though N will not be present, in the hashtable, associated to the same singleton subtree as M. In order to avoid this failure, the state N, and all states that do not have a singleton subtree at P, are placed in a special bucket against which all arriving states are tested for inclusion. This is the `others` field of the `superposition` record.

To sum up, in order to check if a new state is included in a set of previous states `s`, it is sufficient to hash it (by taking the tag of its subtree at P) and check it for inclusion against the states the same bucket of `s.table`, as well as all the states in `s.others`.

Choice of P. A good choice of the path P is paramount to the efficiency of the algorithm. If a P is chosen such that a majority of states have non-singleton subtrees at P, or have the same singleton subtree, the algorithm essentially reverts to the naïve (and quadratic) systematic inclusion test of each new state in all previous states.

The current implementation chooses the path on the basis of the first two states. The chosen path P is simply the first encountered path at which the subtrees are singleton in both the available states, and the subtrees are different between the two available states. This may seem casual considering the importance of choosing a good function π, and improvements could be considered (in particular re-indexation using a new path P' chosen on the basis of all already available states when the initial choice turns out not to work well). The simple version currently implemented works well in practice.

In most loops, the value of the loop index is known precisely and different in each state, so the subtrees that contain only the loop index are often a good choice.

Conclusion

We have formalized an abstract interpreter on graphs that refine the control-flow graph of the input C program, with inspirations from trace partitioning [12] and model-checking [3]. This idea is implemented in TIS-Analyzer, as an improvement over a previous representation of value analysis results as a map from statements to abstract memory states.

The derived analyses that depend on the value analysis' results are in the process of being adapted to take advantage of result graphs. Where this transition has already been made, it has led to noticeable precision gains. Result graphs also enable new derived analyses that were previously difficult to conceptualize, such as a whole-program dependency analysis, with applications when studying software security.

It should be pointed out that thanks to semantic unrolling, the value analysis can be used as an ordinary interpreter: when used to analyze a program's behavior for specific, known inputs, unrolling all loops amounts to emulating the execution from beginning to end (to that end, it helps that internal sources of non-determinism, such as `"foo" == "foo"`, are treated as undesirable errors). In this interpreter mode, the value analysis does not emit any false positives: any warning it emits is for an error that really happens, and in particular that happens for the specific inputs that were used. The result graph allows this local optimality property to be transmitted from the value analysis to the derived analysis built on top of it, so that these analyses too can provide results without false positives when applied to specific inputs.

The result graph is much larger than the control flow graph when the `slevel` is high, but remains within a constant factor of the per-statement storage of memory states that was previously used.

Bibliographie

[1] Erroneous GCC bug report. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475`. Accessed: 2016-08-18.

[2] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI C Specification Language. `http://frama-c.com/download/acsl.pdf`. Version 1.11. Accessed: 2016-08-19.

[3] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*. Springer, 2007.

[4] François Bobot, Sylvain Conchon, E Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008, 2013.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011.

[6] Pascal Cuoq. libksba memory issue disclosure. `http://seclists.org/oss-sec/2016/q3/343`. Accessed: 2016-08-20.

[7] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in OCaml 3.10.2. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 13–22. ACM, 2008.

[8] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 12–19, New York, NY, USA, 2006. ACM.

[9] Atsushi Kanamori and Daniel Weise. Worklist management strategies for dataflow analysis. Technical report, 1994.

[10] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[11] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[12] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.