CARL VON OSSIETZKY universität OLDENBURG

Software-Analyse und Transformation

Tagungsband des gemeinsamen NASA Seminars
der Softwaretechnik-Abteilungen der
Universitäten Bremen, Hamburg-Harburg und Oldenburg

Rainer Koschke (Universität Bremen)
Sibylle Schupp (TU Hamburg-Harburg)
Andreas Winter (Carl von Ossietzky Universität Oldenburg)

mit Beiträgen von

Alexander Galkin (TU Hamburg-Harburg)
Bernd Katzmarski (Universität Bremen)
Hauke Neemann (Carl von Ossietzky Universität Oldenburg)
Christian Neuenstadt (TU Hamburg-Harburg)
Ole Jan Lars Riemann (Universität Bremen)
Malte Schultjan (TU Hamburg-Harburg)
Felix H. Wenk (Universität Bremen)
Arne Wichmann (TU Hamburg-Harburg)

OLNSE Number 1/2010
Sommersemester 2010

Oldenburg Lecture Notes in Software Engineering

Oldenburg Lecture Notes
on Software Engineering (OLNSE)
Carl von Ossietzky University Oldenburg
Department for Computer Science
Software Engineering
26111 Oldenburg, Germany

V die Berechnung von Unterschieden dargestellt, die von dem Werkzeug SiDiff genutzt wird. Anschließend wurden in Abschnitt VI grundsätzliche Visualisierungsmöglichkeiten der Unterschiede von Diagrammen aufgezeigt und die sinnvollste Visualisierung (das einheitliche Dokument) an einem Beispiel detaillierter erläutert. Die benutzerfreundlichste Visualisierung ist sicherlich das einheitliche Dokument.

In Zukunft müssen die Datenmodelle anderer Diagrammtypen für SiDiff aufgestellt werden, damit auch diese miteinander verglichen werden können. Weiterhin wäre es sinnvoll auf einen Standard für die Datenmodelle zurückgreifen zu können, um so die Entwicklung von Werkzeugen zu erleichtern.

LITERATUR

[1] Architecture Board ORMSC, "model driven architecture - a technical perspective," p. 31, 2001.

[2] CollabNet, "subversion," Letzter Zugriff: 14.05.2010. [Online]. Available: http://subversion.tigris.org/

[3] GIT, "git," Letzter Zugriff: 14.05.2010. [Online]. Available: http://git-scm.com/

[4] E. W. Myers, "an o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[5] J. Hunt, K.-P. Vo, and W. F. Tichy, "an empirical study of delta algorithms," 1996.

[6] W. F. Tichy, "the string-to-string correction problem with block moves," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 309–321, 1984.

[7] D. Ohst, M. Welle, and U. Kelter, "differences between versions of uml diagrams," in *ESEC / SIGSOFT FSE*, 2003, pp. 227–236.

[8] P. Lindsey, Y. Liu, and O. Traynor, "a generic model for fine grained configuration management including version control and traceability," in *ASWEC '97: Proceedings of the Australian Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 1997, p. 27.

[9] O. M. Group, "unified modeling language infrastructure," Letzter Zugriff: 30.06.2010. [Online]. Available: http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/

[10] J. Rho and C. Wu, "an efficient version model of software diagrams," in *APSEC '98: Proceedings of the Fifth Asia Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 1998, p. 236.

[11] J. Jones, "abstract syntax tree implementation idioms."

[12] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "xml-ql: a query language for xml." [Online]. Available: http://www.w3.org/TR/NOTE-xml-ql

[13] J. Robie, J. Lapp, and D. Schach, "xml query language (xql)." [Online]. Available: http://www.w3c.org/TandS/QL/QL98

[14] W. W. W. Consortium, "extensible markup language," Letzter Zugriff: 14.05.2010. [Online]. Available: http://www.w3.org/XML/

[15] E. R. Harold and W. S. Means, *xml in a nutshell*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[16] C. Treude, S. Berlik, S. Wenzel, and U. Kelter, "difference computation of large models," in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 295–304.

[17] O. Grassow, "vergleich molekularer graphen mit hilfe des sidiff-algorithmus," Ph.D. dissertation, Universität Siegen, 2008.

[18] J. Wehren, "ein xmi-basiertes differenzwerkzeug für uml-diagramme," Master's thesis, Universität Siegen, 2004.

[19] S. Wenzel, "scalable visualization of model differences," in *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*. Leipzig, Germany: ACM Press, May 2008, pp. 41–46.

[20] U. Kelter, M. Monecke, and D. Platz, "constructing distributed sdes using an active repository," in *University of South Australia, Australia*, 1999, pp. 17–18.

[21] U. Kelter, "h-pcte - a high-performance object management system for system development environments," in *COMPSAC*. IEEE Press, 1992, pp. 45–50.

[22] D. Ohst and U. Kelter, "a fine-grained version and confguration model in analysis and design," in *ICSM*, 2002, pp. 521–.

Introduction to Temporal Path Conditions in Dependence Graphs

Malte Schultjan TUHH

***Abstract*—"Program dependence graphs" are widely used to represent possible information flow in a program. "Path conditions" in dependence graphs are a way to provide conditions for information flow along a path or chop in a dependence graph. Path conditions can be boolean. But then they have the drawback that they can not express temporal conditions. For example that one condition has to hold until another one holds. Therefore Andreas Lochbihler and Gregor Snelting introduced temporal path conditions in their paper "On temporal path conditions in dependence graphs" [1] They have shown that these are an improvement of boolean path conditions by proving the soundness property if a temporal path condition for a path is satisfiable, then the ordinary boolean path condition for the path is satisfiable holds while the converse does not hold. This paper summarizes and introduces their work.**

I. Introduction

In this paper you will find a short overview over everything important to create temporal path conditions. In section 2 the building blocks required to construct temporal path conditions are presented. These are program dependence graphs, path- and execution conditions and slicing to create chops. Static single assignment will be introduced as a way to handle different runtime instances of a program variable while creating execution conditions. Boolean path conditions have some problems, most important here that boolean path conditions are not able to express the fact that some condition must hold after some other condition is fulfilled. Therefore we show in section 3 how temporal path conditions can be created using temporal operators based on Linear Temporal Logic to overcome this problem. At last, some simplification rules will be presented, which can be used to to shrink the size of the algorithmicly created LTL path conditions to one reasonable for automated model checking.

II. Path conditions in dependence graphs

Program dependence graphs are used to show that one program statement has to be executed before another one. In program dependence graphs, nodes are used to represent a program statement, edges stand for either "control" or "data dependences". We first explain and illustrate both terms, then discuss how they are used for "slicing" and "boolean path conditions".

A. Control dependency

A statement "t" is "control dependent" on another statement "s", if it will only be executed after the condition stated in "s" evaluates to a specific outcome. In the following example, "b" is control dependent on "if(a)" and so is "c".

```
1 if (a) then // statement s
2     b;
3 else
4     c;
```

This is written as $s \xrightarrow{a} b$ and $s \xrightarrow{\neg a} c$, because if "a" evaluates to true "b" will be executed, otherwise "c". The transitive and reflexive closure of $\rightarrow$ is written as $\rightarrow^*$ In the following snippet,

```
1 while (a) { //statement s
2     b;
3 }
4 c;
```

"b" is control dependent on "s", but "c" is not control dependent on "s", if we assume that our program always terminates, which means every loop has to terminate at some point.

B. Data dependency

A statement "s" is data dependent on "t" with regard to "x" if "x" is a variable that is changed in "t" and used in "s" without reassignment in between. For example, in the code

```
1 x = 3; //statement s
2 y = x; //statement t
```

"t" is data dependent on "s" with regard to "x", which is written as $s \xrightarrow{x} t$. A data dependence is loop-carried if there is a loop "a" that includes "s" and "t". This is illustrated in the next statement.

```
1 while (a) {
2     x = 3; //statement s
3     y = x; //statement t
}
```

And we say a data dependence is leaving a loop "a" if only statement "s" is inside the loop. In the example $s \rightarrow t$ leaves the loop "a".

```
1 while (a) {
2     x = 3; //statement s
3 }
4 y = x; //statement t
```

C. Slicing

As shown above, PDGs are used to model information flow, either by direct flow via data dependence or indirect as control dependence. Indirect because the value of "a" as condition for

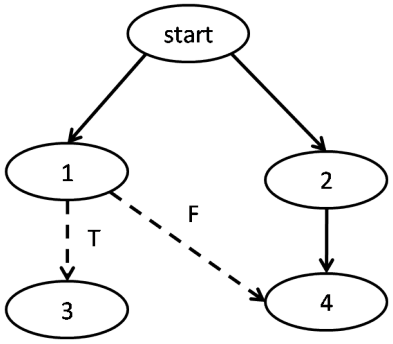

Fig. 1. Slicing example

an if statement can be inferred from the branch of the if or else statement that is executed. A path $\pi : s \rightarrow^* t$ in the PDG means that information flow is theoretically possible from "s" to "t". Slicing can be used to get sets of statements that are influenced by the same statement or influence the same statement. The backward slice $BS(t) := \{s | s \rightarrow^* t\}$ conservatively approximates the set of statements that can influence "t", so if "s" is able to influence "t" it is in the BS. The forward slice $FS(s) := \{t | s \rightarrow^* t\}$ accordingly includes all nodes that "s" can influence. These two can be intersected to get the chop for "s" and "t", $CH(s,t) = FS(s) \cap BS(t)$. For example in Fig. 1, the forward slice $FS(1)$ for "1" is $\{1,3,4\}$ and the backward slice $BS(4)$ for "4" is $\{start, 1, 2, 4\}$; the chop $(1,4)$ would hence be $\{1,4\}$. In Information flow control, slicing can be used to show that computed public values do not depend on secret inputs. For example, if "p" is the public value and the "s" the secret input and "s" is not in BS(p) "s" can not influence "p" and can therefore not be inferred from the value of "p". Physical side channels like timing are of course not included.

D. Boolean path conditions

Slicing can be imprecise because it is only the upper bound of statements that could influence a given statement. For example

```
1 x = 3;
2 if i = 5
3      y = x;
```

line the backward slice of line 3 includes line 1, but really an influence can only occur if $i = 5$ was true. Thus $i = 5$ is a necessary condition such that line 3 is influenced by line 1.

In the following, we distinguish "path" and "execution conditions" which we will need in section 3. A statement "s" influences a statement "t" if during the execution some information generated in "s" is transported to "t" and used there. A path condition for a path $PC(\pi)$ is the necessary condition so that "s" can influence "t" along the path π. A path condition PC(s,t) is the condition that has to be fulfilled so that s can influence t along an arbitrary path. As one would expect, PC(s,t) is just the disjunction of the path conditions for every possible path between "s" and "t". In Fig. 1, with "start" as statement "s" and "4" as statement "t" these are the conditions over the path $start \rightarrow 1 \rightarrow 4$ and $start \rightarrow 2 \rightarrow 4$.

An execution condition for a node "v" is a necessary condition over program variables for "v" being executed. Because control dependence decides whether a node is reached or not, all conditions on at least one path in the control dependence graph, from "start" to node "v" have to be fulfilled for "v" to be executed. This gives the formula

$$E(v) := \bigvee_{\rho : start \rightarrow^* v\, u \rightarrow u' \in \rho} \bigwedge c(u, u')$$

and thus the path condition as the conjunction of execution conditions for all nodes on a path π

$$PC_B(\pi) := \bigvee_{\text{v node in } \pi} E(v).$$

Because of the absorption law for $\vee$, cycles in the control dependence graph don not have to be added multiple times because they could instantly be deleted again by simplification. For the following program code the control dependence for line 4 consists of $1 \rightarrow 3$ and $3 \rightarrow 4$.

```
1 while (i < 5){
2      i++;
3      if (i = 5)
4           a;
5 }
```

The execution conditions for those two are $(i < 5)$ for line 1 and $(i = 5)$ for line 3 giving us

$$E(4) = c(1,3) \wedge c(3,4) = (i < 5) \wedge (i = 5)$$

$(i < 5) \wedge (i = 5)$ seems like a conflict, but in fact i is reassigned between line 1 and line 3. To get results we can use, the program has to be transformed into static single assignment form. In SSA form, every variable occurs at most once on the left-hand side of an assignment. To that end, all variables get an index number that increases every time it occurs on the left side. The previous loop reads as follows in SSA form.

```
1 if (i_1 < 5){
2      i_2 = x;
3      if (i_2 = 5){
4           ...
5      }
6 }
```

For the execution condition we now get

$$E(4) = c(1,3) \wedge c(3,4) = (i_1 < 5) \wedge (i_2 = 5),$$

which is no longer a contradiction.

Whenever there is more than one possibility which instance of a variable might be meant on the right-hand side of an assignment or in a comparison, a ϕ-function is introduced that maps the right relation. In the case of

```
1 i = 1;
2 if (a) i=2;
3 if (i>1)
```

we would introduce $\phi(i_1, i_2)$ to get

```
1 i_1 = 1;
```

```
2 if (a) i_2=2;
3 i_3 =  φ(i_1,i_2  )
4 if  (i_3>1)
```

The SSA transformation is still not enough because the variable can change if the node it is assigned in is executed multiple times. Multiple assignments can happen whenever the variable is inside of a loop. To solve the problem of multiple assignments variables again have to be renamed after a loop-carried data dependence to ensure they are different before and after. Krinke (2003) and Snelting et al (2006) suggest to add ' to the variable, for example i_1'. Still this might not be enough if a loop-independent edge subsumes a loop-dependent edge, meaning one path of the ϕ function is inside a loop. A subsumption of a loop-dependent edge by a loop-independent edge can only occur when a data dependence edge leaves a loop, so the variables just have to be seperated at such edges too.

III. Temporal path conditions

The presented boolean path conditions still have some problems in the presence of loops and loop-carried data dependencies. For example it might be necessary that some loop u terminates for loop v to be executed. In addition, boolean path conditions do not preserve the order of the nodes on the path because the $\wedge$ operator is commutative. This means we can not express that some condition must hold after another condition is fulfilled. As a solution for this problem temporal path conditions based on Linear Temporal Logic have been proposed [1]. In the next sections we will introduce "LTL" operators and explain how to construct "LTL path conditions". We will further demonstrate how to construct these path conditions for a single path and for chops and at the end present some simplification rules.

A. LTL operators

From the following set of operators we will mainly use the Until Operator, but others could be used too, if necessary.

$\mathbf{G}\ \phi$	means ϕ holds always
$\mathbf{X}\ \phi$	means ϕ holds in the next state
$\mathbf{F}\ \phi$	means ϕ holds eventually
$\psi\ \mathbf{U}\ \phi$	means ψ holds until ϕ holds

B. Construct LTL path conditions

To construct LTL path conditions, the first building blocks we need are loop termination conditions L(v). Assuming "u" is a loop predicate node and "v" is not control dependent on "u" but dominates "v" in the CFG, "u" must have been executed and the loop must have terminated before "v" can be executed. This means the negated predicate of "u" must have held at the last execution. L(v) is the conjunction of all those conditions. In the following example the predicate $\neg u$ must hold after the termination of the loop.

```
while(u) {
    ...
}
\\not u must hold
v;
```

The same principle holds true for data dependencies that leave a loop. If a data dependence $e = v \xrightarrow{x} w$ leave the while loop, the negated predicate of "u" must hold at "w". As before, L(e) is the conjunction of all those conditions. The execution condition is called E(v) and contains all those L(v).

C. Path conditions for a single path

To construct a path condition for a single path we will use a recursive algorithm. If the path only consists of one node "s", the PC is the execution condition for "s". Otherwise, "e" is the first edge in the path π with source node "s" and target node "s'", π' is the rest of the path. If "e" is a control dependence, E(s) is added conjunctively to PC(π') (note the recursion).

If "e" is no control dependence the path condition is $E(s) \wedge E(e)\mathbf{U}(L(e) \wedge \phi(e) \wedge PCPath(\pi'))$ where $PCPath(\pi')$ denotes the recursive part.

D. Path conditions for chops

Between two statements there is usually more than one path, but just taking the disjunction of path conditions for all paths between the two statements might lead to infinite formula because of cycles. To deal with cycles, the path is first divided into cycle-free paths. After that a condition is calculated for every path that captures information flow along π and every other path that starts or ends with the same node. At every node "v", an **U** is inserted in the following way: The first operator θ is the execution condition for all nodes in a cycle including v. If no such cycle exists, it is set to false. The second operator is the next recursion step, as before, when it is a control dependence edge added conjunctively, otherwise as $E(s) \wedge E(e)\mathbf{U}(L(e) \wedge \phi(e) \wedge E(s')) \wedge \theta\mathbf{U}(L(e) \wedge PCPath(\pi'))$. The complete path condition for a chop is then the disjunction of all possible paths in the PDG.

E. Simplification

Because of the recursion seen before the PCs grow rapidly even for small programs or chops. But because they typically include a large number of redundant statements, LTL identities can be used to simplify them to a size feasible for model checkers in a realistic amount of time. There is a large number of these identities and at this point only three will be presented to give a general idea. More can be found in the original paper [1].

$$\frac{A \Rightarrow B}{A\mathbf{U}(B\mathbf{U}C) \equiv B\mathbf{U}C}$$

$$\frac{B \Rightarrow A}{A\mathbf{U}(B\mathbf{U}C) \equiv A\mathbf{U}C}$$

$$\frac{C \Rightarrow A, D \Rightarrow B}{A\mathbf{U}(B \wedge C\mathbf{U}D) \equiv A\mathbf{U}D}$$

In these identities, the first line represents the statement that has to be fulfilled so that the equivalence in the second line is true. For the first example this would be: If $A \Rightarrow B$ then $A\mathbf{U}(B\mathbf{U}C)$ can be written as $B\mathbf{U}C$.

IV. Conclusion

Information-flow control based on Program Dependence Graphs today can handle medium sized programs and show for them if information flow between two program points is possible or absolutely impossible. Path conditions are more detailed: if the condition can not be fulfilled no information, flow is possible, otherwise the solution for the condition is a witness for illegal information flow. It can be shown that the extension of boolean path conditions by temporal operators is an improvement. If the temporal path condition is satisfiable the boolean path condition is satisfiable too, the inverse is not valid. As can be seen in the section about temporal path conditions, temporal operators are a useful improvement of boolean path conditions. Model checkers like SPIN or NuSMV can be used to find out if a sequence for information flow between two points is possible. This sequence is interesting for both software security and safety and might be used to show that software is secure within a number of given specifications. Until now, only the theoretical foundations for a limited programming language consisting of if and while statements exist, but still these foundations are an important step on the way to constructing better software.

References

[1] Andreas Lochbihler, Gregor Snelting *On Temporal Path Conditions in Dependence Graphs*, 7th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM September 2007), pages 49–58, url: http://pp.info.uni-karlsruhe.de/uploads/publikationen/lochbihler07scam.pdf

Blockseminar Softwareanalyse- und Transformation: Statische Zeigeranalyse

Ole J. L. Riemann

Zusammenfassung—**Diese im Seminar „Softwareanalyse und -Transformation“, als Ergänzung zu einem Vortrag entstandene, Ausarbeitung soll einen kleinen Überblick über das Thema der statischen Zeigeranalyse geben. Zunächst werden kurz einige Anwendungsmöglichkeiten von statischen Zeigeranalysen beschrieben, um den Leser für das Thema zu motivieren. Nach der Einführung einiger Begriffe wie Alias-Set und Points-To-Map werden die Möglichkeiten zur Klassifizierung von statischen Zeigeranalysen nach Fluss-, Datenfluss-, Feld- und Kontext-Sensitivität vorgestellt und erläutert. Anschließend werden drei verschiedene Vorgehensweisen zur Bestimmung von Zeigerzielen beleuchtet. Zwei schon etwas ältere Analysen stammen von Lars Ole Andersen beziehungsweise von Bjarne Steensgaard und ein neuerer Ansatz von Stefan Staiger. Nach der Erklärung der Verfahren werden sie miteinander verglichen und in die zuvor vorgestellten Kategorien eingeordnet. Abschließend werden unzureichend gelöste Probleme und Erweiterungen, welche die Autoren der Verfahren zur Zeigeranalyse aufzeigen beziehungsweise vorschlagen beschrieben.**

I. Einführung

Für jedes nicht triviale Programm, das in einer Sprache geschrieben ist, die das Konzept der Zeiger implizit oder explizit unterstützt, ist es wichtig zu wissen, auf welche Speicherbereiche oder Objekte die einzelnen Zeigervariablen zeigen. Zum Beispiel will ein Programmierer wissen, ob ein Zeiger, der an einer Stelle definiert wurde und den er an anderer Stelle verwendet, auf *null* oder eine zulässige Speicher-Adresse zeigt. Dies kann man bei kleinen Programmen noch gut manuell in Erfahrung bringen, doch sobald die Programme größer werden und sich über viele Funktionen und Module verteilen, wird es zu einer für Menschen unlösbaren Aufgabe. Neben diesem konkreten Beispiel können statische Zeigeranalysen auch allgemein der Fehleranalyse und der Seiteneffektanalyse dienen. Des Weiteren bilden sie die Basis in einigen Compiler-Optimierungen, wie zum Beispiel der sogenannten „Constant-Propagation“. Oft sind Zeigeranalysen notwendig für nachfolgende Analysen und können zum Programmverstehen beitragen. Schließlich spielen sie auch eine Rolle bei der Architekturrekonstruktion. Ohne eine Zeigeranalyse kann schon das Ergebnis einer simplen Addition zweier Variablen nicht vorhersagbar sein, obwohl bekannt ist welche Werte den einzelnen Variablen zugewiesen wurden, wenn man nicht weiß, ob es sich bei diesen Variablen um Aliase handelt.

Mittels einer Alias-Analyse kann diese Beziehung zwischen zwei oder mehreren Zeigervariablen untersucht werden. In vielen Fällen ist aber nicht nur die Information interessant, ob zwei Variablen auf das gleiche Objekt zeigen, sondern wohin genau sie zeigen oder zeigen könnten. Eine „echte“ Zeigeranalyse, deren Ergebnis eine Menge an Zeigerzielen für die jeweilige Zeigervariable ist, kann diese Informationen liefern. Das Problem der Bestimmung von Alias-Variablen ist dann nur noch eine Spezialisierung einer allgemeinen Zeigeranalyse.

2001 schreibt Hind [5] von über 75 Publikationen zum Thema Zeigeranalyse innerhalb der letzten einundzwanzig Jahren. Heute kann man den Literaturangaben von Staiger [2] schon über zehn weitere zu diesem Thema entnehmen, sodass eine Schätzung von über 100 Publikation angebracht sein dürfte.

Publikationen, wie die von Hind [5] oder von Staiger [2], die einen Überblick über die Thematik geben wollen oder detailliert in das Thema einführen, lassen erkennen, dass alle Zeigeranalysen Attribute gemeinsam haben, nach denen sie klassifiziert werden können. Neben den wichtigsten Möglichkeiten der Klassifizierung soll diese Ausarbeitung auch drei unterschiedliche Verfahren zur statischen Zeigeranalyse beleuchten, sodass sich der Leser einen groben Überblick über das Thema verschaffen kann.

II. Hintergrund und Klassifizierung

Hind [5] grenzt zunächst die Aliasanalyse von der Zeigeranalyse wie folgt ab: *„A pointer alias analysis attempts to determine when two pointer expressions refer to the same storage location. A points-to analysis [...] attempts to determine what storage locations a pointer can point to.“*
Eine Zeigeranalyse berechnet also eine Menge von Zielen, auf die die jeweilige Zeigervariable (oder nach Hind der „Zeigerausdruck“) zeigen kann. Die Ausgabe einer Aliasanalyse für jede untersuchte Zeigervariable ist jedoch eine Menge von Zeigervariablen, welche auf den gleichen Speicherbereich zeigt. Hier wird nicht klar, dass auch eine Aliasanalyse meistens überschätzend ist. Im Gegensatz zur Aliasanalyse basieren Zeigeranalysen auf einer kompakteren Repräsentation der Daten, was folgendes Beispiel von Andersen [1] (S. 113) in Abbildung 1 zeigen soll:
Wie in Abbildung 2 schon erkennbar ist, wird die Ergebnismenge einer Zeigeranalyse (auf der rechten Seite) kompakter als die einer Aliasanalyse (auf der linken Seite).

Essenziell für die Anwendung einer Zeigeranalyse ist, dass die Sprache, in der das zu analysierende Programm

```
int x, y, *p, **q, (*fp)(char *, char *)    1
p = &x;                                     2
q = &p;                                     3
*q = &y;                                    4
fp = &strcmp                                5
```

Abbildung 1. Beispielprogramm von Andersen zur Unterscheidung von Alias- und Zeigeranalyse

Alias-Set	*Points-To-Map*
(*p, x),	p -> {x, y},
(*q, p),	q -> {p},
(** q, x),	fp -> {strcmp}
(** q, y),	
(*fp, strcmp)	

Abbildung 2. Vergleich: Alias-Set und Points-To-Map

geschrieben ist, Zeiger auch unterstützt. Es gibt aber auch Sprach-Features, die eine Zeigeranalyse erschweren. Beispielsweise können Zeigervariable in der weit verbreiteten Sprach C nicht nur auf den Heap, sondern auch auf den Stack zeigen. Zudem werden „multi-level“ Zeiger (also Zeiger auf Zeiger) angeboten und Integer können als Zeiger interpretiert werden. Schließlich kann auch Zeigerarithmetik problematisch sein. Da das Problem der Zeigeranalyse im Allgemeinen unentscheidbar ist, können alle Algorithmen zur Bestimmung von Zeigerzielen die Ergebnisse lediglich approximieren. Alle drei unten vorgestellten Algorithmen sind dabei überschätzend. Es werden also falsch positive Zeigerziele generiert. Zum Vergleich der Verfahren hilft es, wenn man sie kategorisieren kann. Im Folgenden sollen vier Möglichkeiten zur Kategorisierung vorgestellt werden, die Einfluss auf die Präzision der Analysen haben.

A. Fluss-(In-)Sensitivität

Zunächst lassen sich Zeigeranalysen in fluss-insensitive und fluss-sensitive Vertreter unterteilen. Dabei wird Fluss-Sensitivität von Staiger wie folgt definiert:
„Eine Analyse ist fluss-sensitiv (FS), wenn sie den Kontrollfluss beachtet, andernfalls fluss-insensitiv (FI). . . .“([2], S.55)

```
int a, b, *p;                               1
p = &a;                                     2
p = &b;                                     3
```

Abbildung 3. Beispielcode zur Fluss-(In-)Sensitivität

```
[p -> {a, b}]
```

Abbildung 4. fluss-insensitives Ergebnis

Definieren wir uns in einem kleinen Beispielprogramm wie in Abbildung 3 also zwei Integer a und b und einen Zeiger auf einen Integer p und lassen den Zeiger nacheinander auf a und b zeigen, so kann eine fluss-insensitive Zeigeranalyse nicht feststellen, dass die Zuweisung der Adresse von b an p die

```
[p -> {b}]
```

Abbildung 5. fluss-sensitives Ergebnis

vorherige Zuweisung der Adresse von a an p überschreibt und es entsteht eine „Points-To-Map“ wie in Abbildung 4. Eine fluss-sensitive Analyse würde die Aktualisierung sehr wohl feststellen und ein Ergebnis, wie in Abbildung 5 dargestellt produzieren.

Fluss-insensitive Analysen unterscheiden also nicht, in welcher Reihenfolge Instruktionen durchgeführt werden. Es gibt daher im Gegensatz zu fluss-sensitiven Varianten auch keine unterschiedlichen Ergebnisse für verschiedene Programmpunkte. Der Vorteil bei diesem Vorgehen ist jedoch, dass die Zeigervariable nicht pro Verwendung in den Datenstrukturen repräsentiert werden muss. Das Mengengerüst einer fluss-insensitiven Analyse ist also kleiner, was sich nur positiv auf die Laufzeit auswirken kann.

B. Datenfluss-(In-)Sensitivität

Des Weiteren können statische Zeigeranalysen in Bezug auf ihre Sensitivität der Datenfluss-Richtung unterschieden werden. Datenfluss-Sensitivität wird von Staiger wie folgt definiert:
„Eine Analyse ist gerichtet oder datenfluss-sensitiv, wenn sie bei einer Kopierzuweisung die Datenfluss-Richtung beachtet. . . .“ ([2], S.55)
Hind beschreibt Datenfluss-Insensitivität als einen Spezialfall einer fluss-insensitven Analyse, Staiger betrachtet datenfluss-insensitive Analysen nicht genauer. Die Analyse von Andersen unterscheidet sich allerdings genau in diesem Punkt von der Analyse von Steensgard, wie später noch genauer untersucht wird. Im Allgemeinen kann durch Datenfluss-Insensitivität jedoch ein großer Geschwindigkeitsgewinn erzielt werden der allerdings zulasten Präzision geht. Ein Beispiel zu Datenfluss-Insensitivität findet sich im Abschnitt IV.

C. Kontext-(In-)Sensitivität

Jedes nicht triviale Programm, welches in einer höheren Sprache geschrieben wurde, besteht mit Sicherheit aus mehreren Modulen oder zumindest Funktionen. Jede dieser Funktionen kann generell von beliebigen anderen beliebig oft aufgerufen werden und ihre Arbeit so in verschiedenen Kontexten verrichten. Es macht also Sinn, die Zeigeranalysen, die diese verschiedenen Kontexte von Funktionen erkennen von denen zu unterscheiden, die dies nicht tun. Dabei definiert Staiger Kontext-Sensitivität wie folgt:
„Eine Analyse ist kontext-sensitiv (CS), wenn sie die Resultate für ein Unterprogramm in verschiedenen Kontexten unterscheiden kann. Wenn sie nur ein zusammenfassendes Resultat für alle Kontexte liefert, ist die Analyse kontext-insensitiv (CI).“ ([2], S.56)

Zur Veranschaulichung können wir uns also einen Beispielcode wie in Abbildung 6 vorstellen, der eine Funktion definiert, die einen Zeiger auf einen Integer als Parameter

```
void f (int *p) {                    1
  // ...                             2
}                                    3
// ...                               4
int a, b;                            5
f(&a);                               6
f(&b);                               7
```

Abbildung 6. Beispielcode zur Kontext-(In-)Sensitivität

erwartet und mit diesem dann weiterarbeiten möchte. Weiter unten im Code wird die Funktion dann zweimal aufgerufen und ihr jedes Mal verschiedene Adressen übergeben. Eine kontext-insensitive Zeigeranalyse würde die Kontexte der beiden Aufrufe zusammenfassen und dabei in der Funktion f für den Zeiger p eine Zeigerzielmenge wie in Abbildung 7 auf der linken Seite bestimmen. Insbesondere fassen Analysen, welche die einzelnen Kontexte nicht unterscheiden auch die Ergebnisse für Zeigervariablen in Rückgabe-Parametern verschiedener Aufrufe einer Funktion zusammen und liefern an jeden Aufruf-Punkt die gleichen Zwischenergebnisse zurück. Kontext-sensitive Zeigeranalysen unterscheiden die einzelnen Kontexte und würden zu zwei verschiedenen Zeigerzielmengen kommen.

kontext-insensitiv	*kontext-sensitiv*
`[p -> {a, b}]]`	`6: [p -> {a}]]`
	`7: [p -> {b}]]`

Abbildung 7. Zeigerzielmengen für kontext-sensitive und -insensitive Analysen

D. Feld-(In-)Sensitivität

Schließlich können Zeigeranalysen noch Felder unterscheiden oder eben nicht. Staigers Definition dazu lautet:
„Unterscheidet eine Analyse die Felder von Strukturen als eigene (Unter-)Objekte, so ist sie feld-sensitiv. . . . “ ([2], S.56)
Zur Verständnis soll das Beispiel aus Abbildung 8 dienen.

```
struct s {                           1
  int *a;                            2
  int *b;                            3
};                                   4
struct s *p;                         5
int i = 23;                          6
int j = 42;                          7
p->a = &i;                           8
p->b = &j;                           9
```

Abbildung 8. Beispielcode zur Fluss-(In-)Sensitivität

Hier werden eine Struktur s und ein Zeiger p auf diese definiert. Außerdem haben wir zwei Integer i und j, die wir mit unterschiedlichen Werten initialisieren. Schließlich wird in Zeile 8 dem Feld a der Struktur, auf die p, zeigt die Adresse von i zugewiesen. In Zeile 9 wird dem Feld b der gleichen Struktur die Adresse von j zugewiesen.
Feld-insensitive Analysen würden die Struktur s hier als ein ganzes Objekt betrachten und würden in Zeile 8 und 9 eine vollständige Veränderung von p annehmen, wo feld-sensitive Analysen beide Felder unterscheiden und Zeile 8 und 9 nur als teilweise Veränderung von p betrachten würden. Besonders in Programmiersprachen wie C, in denen Zeiger auf Teile von Strukturen üblich sind, sollten feld-sensitive Analysen eine höhere Präzision erreichen als feld-insensitive.

III. Zeigeranalyse nach Andersen

Ein Vertreter fluss-insensitiver, statischer Zeigeranalysen ist die Analyse von Andersen, die er zur Analyse und Spezialisierung von C-Programmen entwickelt hat, wie schon dem Titel seiner Dissertation [1] zu entnehmen ist.

Der Algorithmus basiert auf einzelnen Zeigerzielmengen, die zusammengefasst werden. Andersen unterscheidet dabei zwei Phasen, die „specification“-Phase und den „inference algorithm“ oder die Lösungsphase.
In der „specification“-Phase wird das Problem für den Inferenz-Algorithmus spezifiziert, indem für jede Operation auf Zeigervariablen ein Constraint generiert wird. Die Eingabe für die Lösungsphase ist dann ein Constraint-Graph, der die Beziehungen zwischen den einzelnen Zeigervariablen und Zeigerzielmengen als Kanten darstellt. Dabei ist der Graph nicht frei von Zyklen und es ergeben sich während der Propagation durch die Transitivität der Bedingungen und indirekte Operationen weitere Bedingungen oder, im Falle des Graphen, neue Kanten.

```
int x, y, *p, *q, *r;                1
p = &x;                              2
p = &y;                              3
q = p;                               4
r = q;                               5
```

$$
\begin{array}{ll}
T_p & \rightarrow \{x, y\}, \\
T_q & \supseteq T_p, \\
T_r & \supseteq T_q \qquad\qquad \Rightarrow T_r \supseteq T_p
\end{array}
$$

Abbildung 9. Beispielcode zu Andersens Analyse

In Abbildung 9 sind ein Codeausschnitt und die Bedingungen, die daraus generiert werden könnten, dargestellt, wobei T_x dabei als Zeigerzielmenge von x zu lesen ist. Zu sehen ist hier, dass es sich bei den Beziehungen zwischen den einzelnen Zeigervariablen um Teilmengen-Beziehungen handelt. Wird wie in Zeile 4 p an q zugewiesen, so wird die Menge an Zeigerzielen für p zu einer Teilmenge der Ziele von q. Des Weiteren lässt sich die Transitivität an diesem Beispiel sehr schön illustrieren. Da $T_q \supseteq T_p$ und $T_r \supseteq T_q$ gilt, folgt daraus, dass auch $T_r \supseteq T_p$ gilt. Bezogen auf den Contraint-Graphen, würde so eine weitere Kante hinzukommen.

Experimente zur Laufzeit führt Andersen leider nur an wenigen kleinen Programmen durch, von denen zwei unter 100 Zeilen Code haben und das größte lediglich ungefähr

5000 Zeilen lang ist. Wie er schreibt, wurden die Experimente auf einer Sparc Station II mit 64MB Hauptspeicher ([1], S.145) durchgeführt und keine Analyse hat länger als 4 Sekunden gedauert. Davon ausgehend lassen sich die sowieso nicht sehr aussagekräftigen Ergebnisse nur schwer auf die heutige Situation übertragen. Mehr Informationen liefert da der im nächsten Abschnitt beschriebene Vergleich zu der Analyse von Steensgaard, den Shapiro und Horwitz vorgenommen haben.

Andersen bezeichnet seine Analyse zunächst als „intra-procedural“ also kontext-insensitiv, beschreibt allerdings in Kapitel 4.6 ein Verfahren, um seiner zuvor beschriebenen Analyse mehr Kontext-Sensitivität zu verleihen. Er entscheidet sich dafür, die einzelnen Varianten von Funktionen über einen statischen Aufruf-Graphen zu unterscheiden.

IV. Linearer Ansatz nach Steensgaard

Einen anderen Ansatz zur Analyse von Zeigerzielen verfolgt Steensgaard in [3]. Das datenfluss-insensitive Verfahren ignoriert dabei sowohl die Reihenfolge von Zuweisungen als auch deren Richtung. Bei diesem Konzept der Unifizierung werden zwei Zeigervariablen und deren Zeigerzielmengen bei einer Zuweisung wirklich gleichgesetzt. Dies ermöglicht die Einteilung der Zeigervariablen in Äquivalenzklassen und somit die Verwendung schneller Union-Find Datenstrukturen.

Steensgaard geht so vor, dass sein Algorithmus zunächst jeder Zeigervariable eine eigene Äquivalenzklasse zuordnet. Im weiteren Verlauf werden die Äquivalenzklassen der Zeiger, die im zu analysierenden Programm über eine Zuweisung in Beziehung stehen, zu einer neuen Klasse vereinigt. Schließlich sind alle Zeigervariablen in einer Äquivalenzklasse eingeordnet und die Menge ihrer möglichen Zeigerziele, die durch den Algorithmus bestimmt wurden, ist die Menge aller Ziele der Zeiger, die sich in der gleichen Klasse befinden.

Bei n Variablen in dem untersuchten Programm werden zunächst genausoviele Äquivalenzklassen erzeugt. Die maximale Anzahl von Vereinigungen, die ausgeführt werden können, ist limitiert durch die Anzahl der Äquivalenzklassen. Der Aufwand einer Vereinigung in einer Union-Find-Datenstruktur wird dabei durch die inverse Ackermannfunktion angenähert, welche sehr langsam steigt. Der gesamte Algorithmus hat also eine Komplexität von unter $O(n\ log\ n)$, was nach Steensgaard die bis dahin asymptotisch schnellste, nicht triviale Zeigeranalyse ist.

Die gewonnene Geschwindigkeit wird allerdings mit einer verringerten Präzision bezahlt, wie schon das einfache Beispiel in Abbildung 10 zeigt. Es ist zu sehen, dass die Zeigervariablen p und q bis Zeile 3 noch ihre eigenen Äquivalenzklassen besitzen und sie jeweils entweder auf x oder auf y zeigen. Durch die Zuweisung der Adresse von x an q in Zeile 4 werden die Äquivalenzklassen von von p und q zusammengefasst. Das Ergebnis ist, dass y ebenfalls zu der Zeigerzielmenge von p gehört, obwohl p und y in eigentlich in keiner direkten Beziehung zueinander stehen.

```
int x, y, *p, *q;                                  1
p = &x;                                            2
q = &y;                                            3
q = &x;                                            4
```

bis Zeile 3:

$p \rightarrow \{x\}$,
$q \rightarrow \{y\}$,

ab Zeile 4:

$p \rightarrow \{x, q\} \leftarrow q$

Abbildung 10. Beispielcode zur Zeigeranalyse nach Steensgaard, wie ihn Shapiro und Horwitz [4] zur Veranschaulichung nutzen.

Für den bereits erwähnten Vergleich haben Shapiro und Horwitz die Analysen von Andersen und Steensgaard implementiert und einige Experimente durchgeführt, die sie in [4] präsentieren. Dabei haben sie beide Analysen für die gleichen Programme ausgeführt und die Ergebnisse in Bezug auf Laufzeit und Präzision der Zeigerziele in der Praxis verglichen. Da beide Analysen überschätzend sind, sind die Ergebnisse mit den kleineren Zeigerzielmengen als genauer anzusehen. Die Programme, die untersucht wurden, haben eine Spannweite von 300 bis 24.000 Zeilen Code. Shapiro und Horwitz machen die Beobachtung, dass sich die beiden Algorithmen bei der Analyse der kleineren Programme mit bis zu 3.000 Zeilen Code in der Laufzeit nicht so stark unterscheiden. In einigen Fällen ist da die Analyse von Andersen sogar schneller. Allerdings wurden die Programme in unter einer Sekunde analysiert, sodass man den Differenzen in der Laufzeit nicht so große Bedeutung beimessen kann.
Aussagekräftiger sind dabei die Analysen der längeren Programme, deren Durchführung mehr als nur eine Sekunde in Anspruch genommen hat. Hier ist, bis auf ein paar Ausnahmen festzustellen, dass Andersens Analyse bis zu zehn mal so lange läuft, generell jedoch nur halb so große Zeigerzielmengen produziert wie Steensgaards Ansatz.

V. Kombinierte Analyse nach Staiger

Ein neueres Verfahren zur Bestimmung von Zeigerzielen beschreibt Staiger in seiner Dissertation [2]. Wie er schreibt, hängen Zeiger-, Datenfluss- und Kontrollflussanalysen stark zusammen und beeinflussen einander. Frühere Verfahren haben die Probleme separat betrachtet und die Analysen nacheinander ausgeführt. So wurde zum Beispiel zunächst ein Kontrollfluss-Graph aufgebaut, eine Datenflussanalyse durchgeführt und schließlich eine Zeigeranalyse angestoßen, die von den Ergebnissen vorhergehender Analysen profitiert. Dabei hätte die zuvor ausgeführte Datenflussanalyse jedoch wesentlich genauere Ergebnisse liefern können, hätten ihr die Ergebnisse der Zeigeranalyse zur Verfügung gestanden. Die vorhergehenden Analysen sind also viel zu konservativ überschätzend.

In seinem Ansatz integriert Staiger die drei Probleme und löst sie iterativ gleichzeitig. Dabei setzt die Analyse auf einer Zwischenrepräsentation (oder auch IR für „Intermediate Representation“) des zu analysierenden Programms auf, sodass sie möglichst unabhängig von konkreten Programmiersprachen ist. Ausgehend von der Darstellung in der IR werden zunächst intraprozedurale Kontrollfluss-Graphen erstellt, welche dann zu einem interprozeduralen Graphen zusammengefügt werden. Analog dazu wird ein interprozeduraler Datenfluss-Graph aufgebaut, mithilfe dessen der Datenfluss bestimmt werden kann. Nach dem von Staiger „Graphaufbau-Schritt“ genannten ersten Teil des in einer Fixpunkt-Iteration durchgeführten Algorithmus werden die Zeigerziele, die sich ergeben über den zuvor aufgebauten Datenflussgraphen, von ihren Quellen zu ihrer Dereferenzierung propagiert. In diesem Schritt werden neue Eingaben für den ersten Schritt der Iteration generiert. Um den Algorithmus zu beschleunigen, bespricht Staiger Möglichkeiten um den Graphen, auf dem gearbeitet wird zu vereinfachen, sodass die Größe reduziert wird. So können zum Beispiel Zyklen erkannt und zu Teilgraphen zusammengefasst werden.

Zur Veranschaulichung verwendet Staiger das Beispiel aus Abbildung 11, wobei *(b)* und *(c)* den Graphen in der ersten Iteration vor beziehungsweise nach der Propagierung der Zeigerzielinformationen zeigen, der aus dem Programm *(a)* entsteht. Definitionen kürzt Staiger mit *def*, Verwendungen mit *use* ab. Die Kanten zeigen in Datenfluss-Richtung. Zu sehen ist hier beispielsweise, dass die Definition *def(p,4)* über den *phi*-Knoten zu der im Beispielprogramm darüber befindlichen Verwendung *use(p,3)* „fließt“. Diese Information wird in der Propagierung ausgenutzt und es kann neben *def(q,3)* auch *def(x,3)* erzeugt werden.

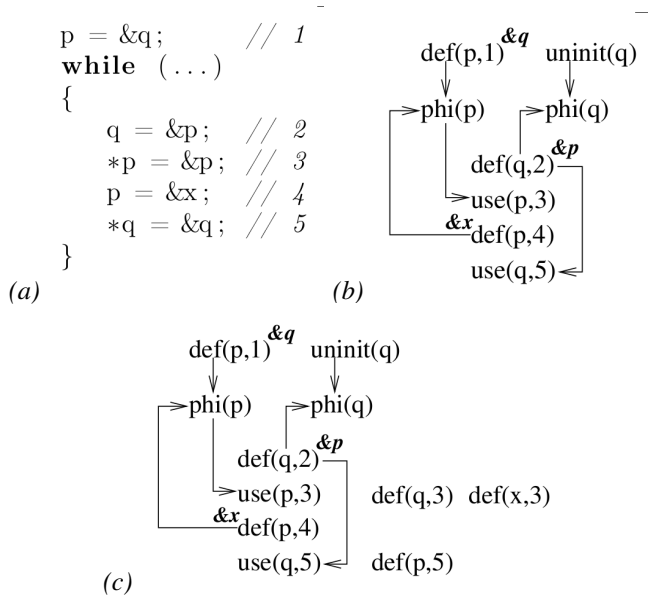

Abbildung 11. Beispiel von Staiger ([2], S.112) zur kombinierten Analyse

Im Gegensatz zu Andersens Analyse ist die kombinierte Analyse von Staiger fluss-sensitiv. In der Dissertation wird allerdings beschrieben, welche Veränderungen vorgenommen werden müssten, um sie fluss-insensitiv auszuprägen. Anstelle einer auf einem Kotrollfluss-Graphen aufbauender Datenfluss-Darstellung würde die Fixpunkt-Interation dann auf einem Graphen ähnlich dem Constraint-Graphen von Andersen arbeiten.

Außerdem ist die Analyse feld-sensitiv, die allerdings davon abhängt, ob Felder bei der Erstellung der IR als separate Objekte modelliert werden. Im Allgemeinen stellt Staiger jedoch eine große Ähnlichkeit seiner Analyse mit der von Andersen fest und hofft, dass sich daher die Beschleunigungen, die bis jetzt für Andersens Algorithmus entwickelt wurden, auch auf seine Analyse übertragen lassen. Zuletzt beschreibt Staiger noch eine Erweiterung, durch die die Analyse Kontext-Sensitivität und somit ebenfalls eine erhöhte Präzision erhält. Allerdings erhöht dies die bisherige asymptotische Komplexität von $O(n^3)$ auf $O(n^4)$.

Im Gegensatz zu Andersen nimmt Staiger neben einer theoretischen auch eine umfassende empirische Evaluation vor. Neben verschiedenen, jedoch ausnahmslos in C geschriebenen, Programmen, deren Größe von 2.000 bis 260.000 Zeilen Code reicht, werden auch die verschiedenen Varianten der Analyse untersucht. Außerdem werden die unterschiedlichen Möglichkeiten der Implementierung verglichen. Die Ergebnisse werden dabei so zusammengefasst, dass durch die Fluss-Sensitivität bei einigen Programmen eine „mäßige“ Zunahme der Genauigkeit bringt, bei wenigen eine „deutliche“. Des Weiteren ist zu beobachten, dass der theoretisch ermittelte, kubische *worst-case* nicht erreicht wird und die Analyse eher ein quadratisches Laufzeitverhalten zeigt. Dabei reicht die reale Dauert von unter einer Sekunde für das kleinste untersuchte Programm bis zu über eine Stunde für das größte Programm in den Experimenten.

VI. Zusammenfassung und Schluss

Abschließend wäre ein Vergleich aller drei vorgestellten Analysen angebracht. Da ich, wahrscheinlich auch aufgrund der Aktualität der Kombinierten Analyse von Staiger, hierzu keine Veröffentlichungen finden konnte muss sich dieser Vergleich auf das beschränken, was ich den Publikationen entnehmen kann.

Wie schon erwähnt untersucht Staiger bereits die Verbindung zwischen seiner Analyse und der von Andersen. Dabei wird klar, dass beide Verfahren in ihrer Grundversion eine kubische, asymptotisch Komplexität der Laufzeit besitzen. Sowohl bei der empirischen Evaluation von Staiger und Andersen als auch bei den Experimenten, die Shapiro und Horwitz [4] durchgeführt haben ist erkennbar, dass die Laufzeit für die untersuchten Programme noch akzeptabel ist. Andersens Versuche an lediglich fünf Programmen mit vergleichsweise kurzen Quelltexten sind dabei jedoch

	Andersen	Steensgaard	Staiger
datenfluss-sensitiv	ja	nein	ja
fluss-sensitiv	nein	nein	ja (optional ohne)
kontext-sensitiv	optional	nein	optional
feld-sensitiv	nein	nein	optional
Aufwandsklasse	$O(n^3)$	$O(n\alpha(n,n))$	$O(n^3)$ bis $O(n^4)$

Tabelle I
TABELLARISCHE KATEGORISIERUNG VORGESTELLTER ZEIGERANALYSEN

weniger aussagekräftig. Die Experimente von Shapiro und Horwitz geben da eine genauere Vorstellung zu dem Verhalten des Verfahrens in der Praxis. Wie beschrieben, zeigt der Vergleich, dass Steensgaards Verfahren bei größeren Eingaben wesentlich schneller ist als das von Andersen. Damit liegt der Schluss nahe, dass es auch schneller ist als die neue, kombinierte Analyse. Die Laufzeiten der Experimente von Staiger sind leider nur schwer mit den älteren Ergebnissen vergleichbar, da er diese auf Hardware durchführt, die um einige Generationen moderner und schneller ist. Auf der Andersen und Steensgaard zur Verfügung stehenden Technik hätten sie wahrscheinlich nicht in vertretbarer Zeit durchgeführt werden können.

Mit mehr Sicherheit kann man allein aus den Experimenten von Staiger schließen, dass die Präzision der Ergebnisse gegenüber denen Andersens Analyse leicht ansteigt, da er seine fluss-insensitive Ausprägung, die der Analyse von Andersen ähnelt mit seiner fluss-sensitiven vergleicht. Da Shapiro und Horwitz gezeigt haben, dass Andersens Analyse präziser aber auch langsamer ist als die von Steensgaard, kann man bestätigen, was eigentlich schon offensichtlich ist: Von der Analyse von Steensgaard über die von Andersen bis zum Verfahren von Staiger nimmt sowohl die Laufzeit als auch die Präzision zu. Staiger erwähnt außerdem noch eine Analyse von Wilson, die genauer zu sein scheint. Allerdings bezeichnet er sie auch als „nicht skalierbar“. Diese Analyse könnte das nächste Element in dieser Reihe bilden.

Betracht man die Klassifizierung der einzelnen Analysen in Bezug auf die zuvor vorgestellten Möglichkeiten, wie sie in Tabelle I dargestellt ist, so sieht man deutlich, dass der Gewinn an Präzision dem „Zuwachs an Sensitivität“ geschuldet zu sein scheint. Die höhere Präzision bei Beachtung der Datenfluss-Richtung ist offensichtlich. Der Präzisionsgewinn bei Fluss-Sensitivität zeigt sich in Staigers Experimenten während Hind 2001 jedoch noch schreibt: *„Empirical studies [. . .] suggest that for context-insensitive analyses, a flow-sensitive analysis does not offer much precision improvement over a subset-based flow-insensitive analysis.“*([5])

Neben der Präzisionssteigerung, die Staiger erreicht, gibt es jedoch noch einige Probleme, die seine Analyse nicht beachtet. So schreibt er, dass die Möglichkeit des Auftretens von Exceptions ignoriert wird. Eine Unterstützung für dieses Konzept könnte jedoch wünschenswert werden, da es in vielen Sprachen verbreitet ist und sogar in C simuliert werden kann. Außerdem geht seine Analyse davon aus, dass Programme sequentiell und nicht parallel abgearbeitet werden. Auch hier scheint es lohnenswert, die Analyse in die Richtung zu erweitern, da Nebenläufigkeit auf zunehmend verbreiteten Multi-Prozessor-Systemen immer mehr an Bedeutung gewinnt und wahrscheinlich immer mehr Programme diesen Umstand ausnutzen wollen. Interessant wäre auch Nebenläufigkeit in der Analyse selbst, wie im Seminar angemerkt wurde. Staiger stellt dazu Überlegungen an und vermutet, dass sich die Analyse sowohl im Graphaufbau-Schritt als auch im Propagierungsschritt parallelisieren ließe, der Grad der Parallelisierbarkeit jedoch von der Eingabe abhinge.

Nach Staiger kann man seine Analyse „ohne größere Schwierigkeiten“ dahingehend verändern, dass sie objektorientierte Programme analysieren kann. Hier wird klar, dass die Unterstützung weiterer Programmiersprachen nicht lediglich eine Sache des Front-Ends ist, das die IR erzeugt. Alle betrachteten Analysen sind in ihrer ursprünglichen Form dazu gedacht, C-Programme zu analysieren, und alle empirischen Experimente wurden auch an C-Code durchgeführt. Überträgt man eine Zeigeranalyse auf Java-Programme kann beispielsweise ausgenutzt werden, dass alle Zeigervariablen nur auf den Heap zeigen können, wie Hind [5] schreibt.

Hind und Staiger sind sich darin einig, dass bedarfsgetriebene („client-driven“) Zeigeranalysen einen Laufzeitgewinn versprechen. So skizziert Staiger eine „Adaptive Analyse“, die sich nicht für jede Eingabe gleich verhält sondern sich entsprechend der Struktur und der verwendeten Sprachmittel in dem zu untersuchenden Programm parametrisieren lässt. Dabei stellt er sich auch vor, dass einige Programmteile als wichtiger markiert werden könnten als andere, welche dann genauer analysiert werden würden. Die implementierte Analyse besitzt dabei bereits die Möglichkeit der „Rückwärts-Propagierung“, bei der Informationen über Zeiger nicht von ihrer Quelle zu ihrer Verwendung sondern in umgekehrter Richtung fließen. Dieses Vorgehen könnte in Analyse-Tools benutzt werden, indem der Benutzer Zeigervariablen markiert, deren Werte ihn interessieren.

Zuletzt muss ich feststellen, dass ich das ganze Thema der statischen Zeigeranalysen aus einer Sicht wahrnehme, die stark von Staigers Dissertation [2] beeinflusst wird. Die große Zahl der Publikationen zum Thema und die Tatsache, dass alle Veröffentlichungen, die dieser Ausarbeitung als Basis dienen auch von Staiger referenziert werden, legt die Vermutung nahe, dass hier einige Aspekte komplett unerwähnt geblieben sind. Mit dem Hintergrundwissen das diese Ausarbeitung vermittelt, wäre die Anwendung einer konkreten Implementierung einer Zeigeranalyse auf einen realen Quelltext und die dabei entstehenden Ergebnisse interessant.

LITERATUR

[1] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language*, DIKU, University of Copenhagen, Dissertation, Mai 1994.

[2] S. Staiger-Stöhr, *Kombinierte statische Ermittlung von Zeigerzielen, Kontroll- und Datenfluss*, Dissertation Universität Stuttgard, 2009.

[3] B. Steensgaard, *Points-to analysis in almost linear time*, Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1996.

[4] M. Shapiro, S. Horwitz, *Fast and Accurate Flow-Insensitive Points-to Analysis*, In: POPL ´97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, S. 1-14, 1997

[5] M. Hind, *Pointer Analysis: Haven't We Solved This Problem Yet?*, In: PASTE '01: 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, S. 54-61, 2001

DIVINE: DIscovering Variables IN Executables
Gogul Balakrishnan and Thomas Reps
University of Wisconsin

Arne Wichmann
Institute for Software Systems
Technische Universität Hamburg-Harburg
arne.wichmann@tu-harburg.de

Abstract

Traditional approaches of discovering variable-like entities in executables do not yield information about indirectly addressed variables. DIVINE provides a way to analyze the possible values of memory locations and thereby allows the analysis of indirectly addressed memory. This paper summarizes the core ideas of the DIVINE algorithm.

1. Executables

Why analyze Executables?

This work presents the methods used in the DIVINE (DIscovering Variables IN Executables) analysis to give precise information about the variable-like entities that can be found in executables.

There are several possible users of such an analysis, including the intelligence community, anti-virus companies, hackers of all shades, computer emergency/incident response teams (CERTs) and common reverse engineers. All of them want to analyze executables directly but have very different reasons to do so.

The simplest case is the unavailability of the source code of a particular executable. This might simply be lost code, which the reverse engineer has to recover, or the analysis of malware, when analyzing the workings of an exploit or the intentions of the author, or the components of an operating system when looking for possible bugs and exploits, just to name some possibilities. Another motivation to analyze executables is the mistrust in the tools used in the development process of an application. This might be the introduction of new and hidden functionality by some component used in the construction process or the analysis whether there is any unintended functionality.

The analysis of executables is also interesting because in the final executable, there usually is a significantly reduced number of paths in the control-flow graph compared with the original source code.

The given intentions for the analysis imply that all kinds of software is analyzed on executable level. Obviously, this is malware, as well as Commercial Of The Shelf (COTS) software, but also in-house software in case of mistrust in the tools used.

To describe the functionality of an executable, information about the control-flow and variable access is needed. When analyzing executables it is relatively easy to give information about directly addressed variable access or program jumps. The frame-relative addressing on the stack is also fairly easy to handle. Indirect addressing of memory is a problem, because it is not trivial to know what the contents of a register or of a memory location may be that is used for addressing.

DIVINE discovers variable-like entities in executables. It does not give information about the program's control flow. It can provide information about indirect jumps and indirect function calls.

Nearly all of this information could be extracted from the executable debug information, but the executable may be stripped of this information or the debug information might be untrusted, as it might be intentionally describing another behavior as the actual code is implementing.

1.1. x86 Assembler

To understand the DIVINE analysis it is necessary to understand the basic concepts of an assembly language. This work uses the x86 assembly language because it is widely deployed and the current DIVINE implementation (Codesurfer/x86) can only analyze x86 executables at the moment. The concepts, however, are of general nature and can be applied to other assembly languages as well.

First, it is described what kind of addressing is possible. Second, the typical locations of variables in the memory are discussed.

Analysis for Event-Driven Programs

Christian Neuenstadt
TUHH

***Abstract*—All provided informations are related to the work of Ranjit Jhala and Rupak Majumdar done in the paper "Interprocedural Analysis of Asynchronous Programs" [1]. Both developed an algorithm for analysing event-driven programs. This paper presents the algorithm and its main ideas.**

Data-flow analysis is a technique for gathering information about the possible set of all values and variables at various points in a computer program. To determine variables and values, which can be reached during execution, a programs control flow graph (CFG) or supergraph is used. For the use of event-driven or asynchronous programming a standard CFG is not able to represent all valid executions, because procedure calls are no longer executed immediately, but can be executed at arbitrary times. In the related work an algorithm is presented to compute the meet-over-valid-paths solution in the given context.

***Index Terms*—Data-flow analysis, asynchronous programs, meet-over-valid-paths solution**

I. Introduction

A. Asynchronous programming

For programming concurrent interactions between different components or clients, asynchronous strategies are very common and can be used efficiently. Compared to a traditional synchronous programming style, where functions are called immediately and the caller has to wait for a return at the callsite, in asynchronous programming every call is first stored in an asynchronous task queue. The queue itself is then controlled by an application-level dispatcher, which executes each call at a later time during the runtime of the main program.

The advantages of asynchronous programming are at hand. Expensive tasks can be delegated and executed at a later point. for event-driven design it is also very important, that different units can wait for asynchronous and external events to interfere the actual execution. If atomicity is ensured, asynchronous executions can run in parallel on multiprocessors or threads.

While there are advantages for software design, on the other side analysing asynchronous software becomes more difficult. The loose coupling of call on one hand and execution on the other makes the program difficult to follow and it becomes harder to write correct programs. For finding bugs and errors especially in this field it is important to analyse code and data flow.

The formalisation of dataflow analysis for synchronous programming has been done by Reps, Horwitz and Sagiv [2] with so called *Interprocedural Finite Distributive Subset* (IFDS) instance. They handled synchronous programming as a graph reachability problem and used a meet-over-all-valid-paths algorithm (MVP) to find the complete set of possible variables and values in a computer program. As in our case asynchronous calls are added, the normal interprocedural control flow graph is no longer able to handle programs and the MVP-algorithm has to be redesigned as well. Jhala and Majumdar have therefore developed *Asynchronous Interprocedural Finite Distributive Subsets* (AIFDS) which can deal with asynchronous programming.

In the following we will use an asynchronous control flow graph to explain valid paths and the use of data flow facts. Later, we summarise the main ideas of the algorithm and show how counters and approximations are used to build the needed meet-over-all-paths.

```
global request_list *r;

main( ) {
  ... //setup request list
  async handle_request( );
}

handle_request( ) {
  If( r == NULL ){
   async handle_request( );
    return;
  }
  rc = malloc(...);
  if( rc == NULL ){
    return OUT_OF_MEM
  }
  async client( rc, r ->);
  r = r -> next; id
  reqs( );
}

Client( client_t *c, int id ){
c -> id = id;
... //client handle
}
```

Fig. 1. Asynchronous example

II. An asynchronous program

In Figure 1 we have a very small client handler related to the example by Jhala and Majumdar. Asynchronous calls are highlighted by a filled grey elipse and synchronous calls just with an unfilled ellipses. The code snippet performes a very simple task. First it sets up the incoming request into the global request list and starts asynchronously the request handler. The

handler checks whether it has a request in its list or not. If not, it starts itself again asynchronously to do any handling at a later time. Otherwise, it tries to allocate some memory for the client and issues another asynchronous call, this time to the client function with a pointer to the reserved memory and current client -id. After that, it sets the pointer of the global request list to the next request and starts itself again. Finally, the client function receives the pointer to the struct and sets it to the given id.

There are two critical points in the program. First, when *client* starts executing and the local variable *c* may not be null. This is ensured by a check in the *handle_request*, which makes sure that the program does not run out of memory. The second critical point is related to the asynchronous structure. It is not guaranteed that the global variable *r* is not null at the point when "'client"' is executed as the coresponding pointer is changed in the function handle_request.

In addition, programs migth be incorrect because an asynchronous procedure might have unexpected side effects on global variables, because it may no longer have the same value when it is executed compared to the time when it was called.

Data flow analysis of synchronous problems is colloquiral done on control flow graphs. We will now formalize a graph, that is able to handle asynchronous problems.

III. Analysing asynchronous problems

A CFG (*Control Flow Graph*) consists of nodes and edges. The representation of a CFG is done by the set (V_p, E_p) where *V* corresponds to the set of particular nodes and *E* to the set of directed edges of the procedure *P*. In asynchronous control flow graphs (ACFG) we have three different kinds of edges:

- *An operational edge* corresponding to a basic block of assignments
- *A synchronous call edge* to a procedure p
- *An asynchronous call edge* to a procedure p

Figure2 shows the ACFG for the example in Figure 1. All control nodes are connected with edges. Solid directed edges are operational edges and directed dashed edges are asynchronous or synchronous calls. There are two special kinds of control nodes, the *start node* and the unique *exit node*. *Start nodes* are at the end of *call-to-start edges* and highlighted with an small arrow at the top. *Exit nodes* symbolize the start of an *exit-to-return edge* and are highlighted by a double circle. Again, asynchronous calls are marked with filled grey boxes and synchronous calls are highlighted by an unfilled grey box.

As asynchronous calls are not executed immediately this picture is not complete. We have to consider the asynchronous task queue with the help of the dispatch procedure "main()" and the special *dispatch node*, which symbolises the application level dispatcher and is able to execute a procedure that has been called several steps before. The *dispatch node* is part of the main function and marked with a filled grey inner circle.

In the ACFG, we can follow paths symbolized by a sequence of control nodes. For example if we follow the path of the main function, we reach at some point the asynchronous call to the function *handle_request*. At a later point, we reach the dispatch node, which stands for a dispatch loop. The application level dispatcher can then execute the pending call waiting in the task queue. During the execution of *handle_request* new asynchronous calls are moving into the task queue and are handled by the dispatcher later at arbitrary times.

In software analysis of asynchronous programs we will need an meet-over-all-valid-paths algorithm. We need only valid paths, because we want to compute only paths that are actually used during the execution of the program. However, we have to find all interprocedural valid paths first.

IV. From ACFG to valid paths

A path is defined *valid*, if it keeps to the language of perfectly balanced parentheses. Intuitively this means that whenever a procedure is executed synchronously both start and exit nodes have to be reached at some point in the program. Moreover, if another procedure is executed, the first procedure has to remain on a stack for later execution. But this is only the definition for synchronous valid paths. When it comes to asynchronous control flow graphs, then we see paths that are still not valid. If we consider the path of the control nodes V_1, V_2, V_3, V_4 and V_{12} in Figure 2, we see that the function *client* is executed here, although there has never been a call to this function before. This path would be invalid. Though, we have to ensure that executions can only take place in connection with a corresponding call in the past. So, during the execution of our programm we will have several pending calls waiting in the task queue and an application level dispatcher, which chooses pending calls and starts their execution. There are several ways and orders in which these execution can take place.

V. scheduling

A scheduling is a mapping between calls on one side and executions on the other. One example is shown in a simplified path in Figure 3. For readability, all nodes that are not calls or executions have been left out. In this example, we can see that for the same sequence two different schedules, and, in that case, two different mappings, are possible. The mappings are shown by the arrows on the left and right side. They connect the filled boxes of the executions with the unfilled boxes of the corresponding calls. Although, we have now restricted the analysis of an ACFG by valid paths and scheduling, there is still a critical need to solve the problem of pending calls, which can completely alter global variables between call and execution. The key challenge is now to find a way for handling an unbounded set of pending calls. The notion of *data flow facts* proves essential here.

VI. Data Flow Facts

Data flow facts are assertions, which can be true or false at a program point. Consider as example the variable *r* (requestlist) from Figure 1. About that variable we possibly know at some point, that it is maybe null or we definitly know that it is not null. At the begin of the mainfunction we know nothing

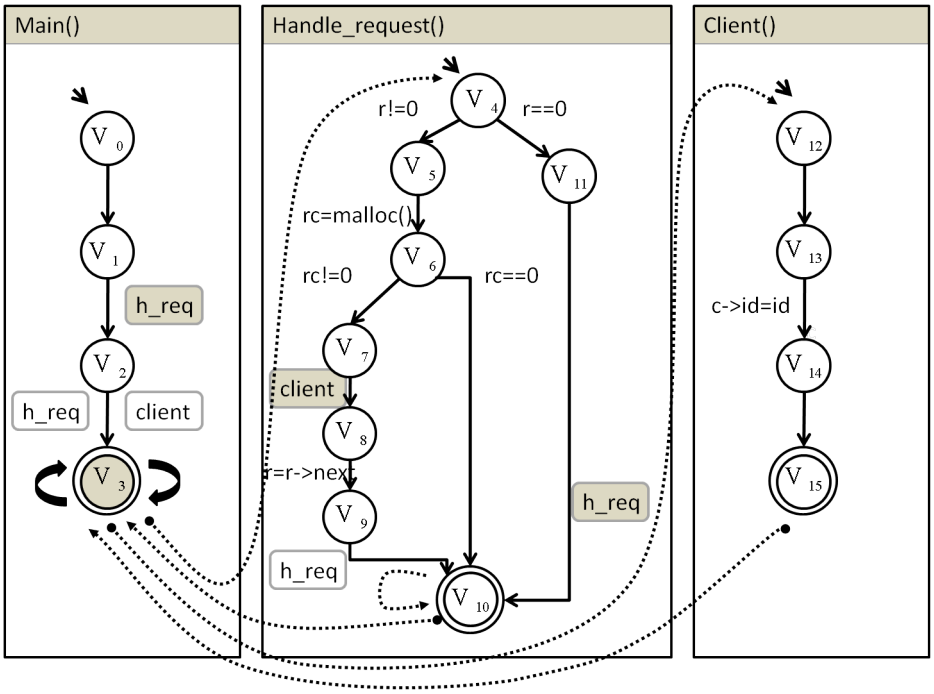

Fig. 2. The ACFG of example1

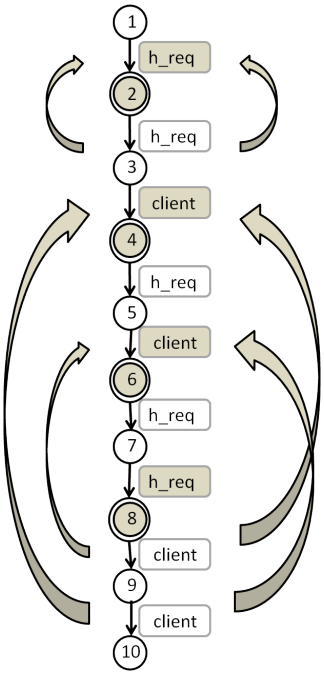

Fig. 3. Two possible schedulings

about the global variables, so we say it is *maybe null*. As we reach the *handle_request function*, there is a check for the global variable *r* and after it, we know whether it is null or not. Therefore if our path says the variable is not null, we can switch our data flow fact to the state *r not null*. But with several asynchronous calls and executions, these global variables can change from null to not null and back by asynchronous calls. While global variables can change, local variables can not. Therefore, we split *data flow facts* into two components: *local data flow facts* and *global data flow facts*.

With two kinds of *data flow facts*, the IFDS instance of synchronous analysis has to be redesigned to an AIFDS instance and looks now as following:

$$A = (G^*, D_g, D_l, M, \sqcap) \tag{1}$$

In the representation A of the AIFDS instance we use the following four components.

1) G* is the Graph of an asynchronous program consisting of V* and E*.
2) D_g and D_l are finite sets of *data flow facts*.
3) M is a function that maps each edge of G* to a distributive data flow transfer function.
4) $\sqcap$ stands for the meet operater.

The approach of solving the problem of several unbounded and pending calls Jhala and Majumdar was to reduce the AIFDS Instance to a standard IFDS instance by encoding the pending calls into the set of *data flow facts* and taking the product with another set, which counts the number of pending calls to an asynchronous function. As this new set of facts is infinite it is only possible to count abstractly, which leads to a finite instance and makes it possible to use the standard algorithm of the IFDS instance.

VII. Algorithm

As we try to encode the pending calls in the IFDS instance to make use of the MVP algorithm by Reps, Horwitz and

Sagiv[2], we have to make changes to our former AIFDS instance. The key step is to use *counter maps* as *dataflow facts*, which results in the following *C-reduced IFDS instance*.

$$(G_C^*, D_C, M_C, \sqcap_C) \tag{2}$$

Instead of global und local *data flow facts* we now use the set of pairs (d, c), which are encoded in D_C. In that pair, d stands for the *data flow fact* and c for the *counter map*, which tracks the number of such calls that are pending. We call this new instance the C_∞*-reduced instance*.

To handle the infinitely large number of data flow facts, we try to build a finite number of modified IFDS instances, which are able to approximate the actual AIFDS instance. We are doing this with an under- and an overapproximation. For a parameter k we call the under-approximation C_k*-reduced* instance and the over-approximation C_k^∞ *instance*.

The parameter k can be chosen. It limits the way of counting for each C_k- and C_k^∞- instance. If the count of a specific pending call reaches the value *k*, the counting stops. The call *k+1* will not be added to the counter. After the *k* calls have been dispatched, the counter has reached zero again. If there is then a dispatch for the *k+1*-th call, the counter is *0* and the path function returns $\perp$. No more pending calls will be executed and the number of pending calls thus reduced to *k*. Overapproximation works similar to the underapproximation. The difference is the counter that is set to infinity when reaching the value *k*. Though, there is always a pending call waiting in the task queue to be executed.

An example of the counting method is shown in Figure 4. It is taken from the example in the previous figure. On the left side, we use a $C_1 - reduced$ and on the right side a C_1^∞ instance. Since *k* equals one in the underapproximation the second call is not counted and in the overapproximation the counter is set to infinity as the second pending call appears.

On the left side, we see that the call at step 6 is dropped and will not be executed later on the path. Finally, the path function returns $\perp$. With the missing call this result is obviously an underapproximation.

On the other side, no call is dropped. Instead, the counter is set to infinity as the second asynchronous call to the client function. At the end, the analysis returns an infinite number of asynchronous executions. Even more, than have ever been called. In other words, we observe an overapproximation.

As the counters C_k and C_k^∞ are finite, we are able to compute the MVP solution for both I_k and I_k^∞ IFDS instances. It can be shown, that I_k is an underapproximation and I_k^∞ an overapproximation of the real AIFDS instance. With an increasing value *k*, we can narrow both approximations and we will reach a point, where the result does not change any more and the two approximations coincide. As both approximations are always under- or over approximations each, the exact solution of our AIFDS instance is reached and the algorithm is guaranteed to terminate.

The algorithm used by Jhala and Majumdar is called ADFA algorithm and shown in Figure 5. The auxiliary RHS algorithm

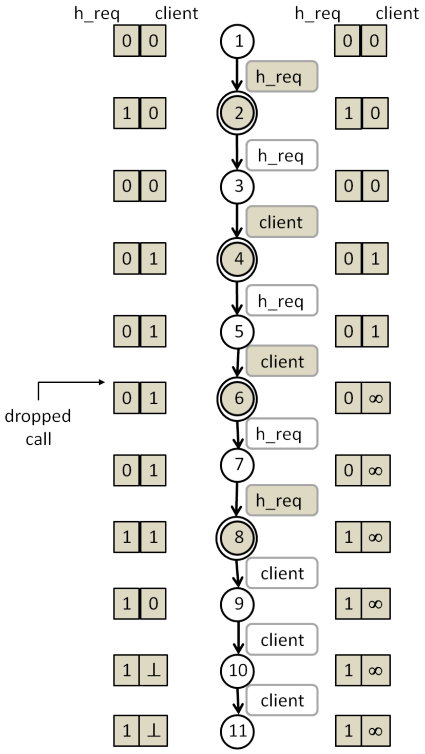

Fig. 4. Counting scheme of an underapproximation (left) and overapproximation (right)

Input: AIFDS instance *A*
Output: MVP solution for *A*
k = 0
repeat
k = k + 1
I_k = C_k-reduced IFDS instance of *A*
I_k^∞ = C_k^∞-reduced IFDS instance of *A*
until RHS(I_k) == RHS(I_k^∞)
return RHS(*Ik*)

Fig. 5. **Algorithm1**: ADFA Algorithm

is the algorithm for finding the meet-over-all-valid-paths created by Reps, Horwitz and Sagiv. [2] We see that the exact AIFDS solution is found as the solution for the over- and underaproximation coincides.

VIII. CONCLUSION

The summarised algorithm provides an procedure for static analysis of asynchronous programs and improving their performance and reliability.

Further informations about the concept of data flow analysis for asynchronous programming and the ADFA algorithm can be found in the original paper by Ranjit Jhala and Rupak Majumdar. [1]

REFERENCES

[1] R. Jhala and R. Majumdar *Interprocedural Analysis of Asynchronous Programs*, POPL 07, ACM, 2007

[2] T. Reps, S. Horwitz and M. Sagiv *Precise interprocedural dataflow analysis via graph reachability*, In POPL 95, ACM, 1995

Improve Security of Web Applications with Taint Propagation

Bernhard Katzmarski
University of Bremen
Computer Science
FB3 - TZI
Bremen, Germany, 28359
Email: bkatzm@tzi.de

***Abstract*—Improper input and output validation is the reason for most security failures today. Especially web applications suffer because they are remotely accessible by everybody.**

If an attacker can insert data which was not expected and is not proper validated, he can compromise the system and harm its users.

This paper explains two common vulnerabilities. SQL Injections and Cross Site Scripting attacks. It is explained why the reasons for these vulnerabilties are always validation failures.

To find improper validations, it can be used a static analysis called taint propagation. Especially in already deployed applications which most of the time are not developed with security in mind, these analysis can point out possible attack vectors which in tradional code reviews might be overlooked.

It summerizes two approaches for finding these vulnerabilities within code. One for Java and one for PHP as representative languages used for todays web development. Apart from the fact that PHP like most other languages used for web applications is dynamically typed, the approaches are very equal. It is always a flowproblem trying to find tainted data that can reach sensitive functions.

I. Introduction

Security leaks seem to be as old as time itself. It is hard to find them with traditional code reviews because it requires a lot of architecture, coding knowlege and especially expertise on security. It has been made a lot of effort to find these vulnerabilities with static analysis.

Good old Buffer Overflow and Format String vulnerabilities are examined very well and can be found with static analysis. (1)(2)

As web applications became popular, some new kind of attacks were exposed. But from a technical viewpoint the reasons for these attacks were quite the same. Improper or missing validations of user input are still the base for a whole lot of attacks.

The great problem web applications have in the opposite to normal programs is, that they are accessible by everybody at everytime from everywhere.

"Traditional defence strategies such as firewalls do not protect against Web application attacks, as these attacks rely solely on HTTP traffic, which is usually allowed to pass through firewalls unhindered." (3)

Attackers try to provide input to applications that is accidentally interpreted as a command for getting the application server to do unexpected things. With strong validations checking for innocent input these flaws can not occur. But since development cycles are getting shorter, web applications are deployed very fast. Nearly every application out there might have at least one vulnerability. The tricky point is that most of the time developers are not aware of their vulnerable parts in code until their application was exploited by somebody.

Thats where static analysis like Taint Propagation, which is described in this paper, comes into play. The main advantages of static analysis like this are that they can be used over and over again during the development cycle. After having an analyser setup it requires nearly no effort to run it. Even previously fixed vulnerabilities which were opened again due to a side effect while developing can be detected.

BK
July 05, 2010

II. Security

Todays software has to meet a lot of requirements. It has to be easy to use, fast, scalable and of course it has to be secure. These are common keywords often used to describe software. Especially companies promote their products to be secure by highlighting features like strong encryption modes, password protection or access control.

But according to Brain Chess (4) it is important to understand the difference between security features and secure features.

The best encryption algorithmus or the best password protection does not help anything if they are not implemented correctly.

"Analyzing reports of security attacks quickly reveals that most attacks do not result from clever attackers discovering new kinds of flaws, but rather stem from repeated exploits of well-known problems." (2)

III. Web Applications

The world wide web is still growing. New web applications and web services are released nearly every day. Everybody is using the web for shopping, banking, chatting, mailing and the newest trend is using these services on mobile devices. Even desktop applications are moving to online plattforms if you think of google calendar for example.

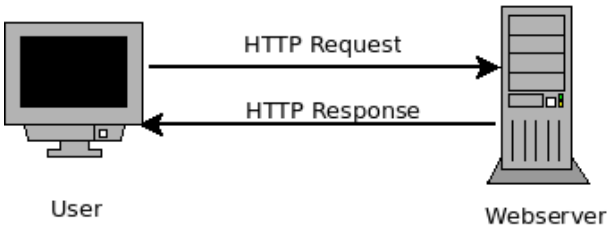

Fig. 1. How an Web Application work

From the developers point of view, all web applications have several things in common. All web applications deal with HTML and HTTP. As the www was developed about 20 years ago, it used only static HTML pages which were connected together with hyperlinks. Today HTML is dynamically generated with a server side script and changed with Java-Script and Ajax in the clients browser to achieve this kind of dynamic user interaction we call Web 2.0.

No matter how much user interaction and dynamic content a web application deals with, it is always the good old client server architecture. The webserver provides several resources which can be accessed by the clients webbrowser by knowing the URL.

The basic setup of a web application typically consists of an apache webserver which provides some modules to delegate incoming requests to a server side script such as php, asp, perl and lets not forget newcomer like ruby on rails. Despite of which scripting language is used to generate the resulting HTML, it is always the same way. Additionally in most cases the script communicates with a database to select or insert data. The result of the script is always HTML which is send back to the client which has requested the page.

Compared to traditional desktop applications, there are some aspects which makes web applications very special. Normal applications have one main method as an entry point and run until they are finished or the user requests them to close.

Instead, web applications do have numerous entry points and if we talk about runtime there is only the runtime between the examination of the request until sending the corresponding response back.

And it becomes even worse due the fact that HTTP is stateless. Historically, there was no need to know something about the user who requests a page. But as web applications came to life, they needed to provide custom content. The well known keywords are registration and authentication or simply login. The user logs in and then the web application knows what he is allowed to do. But what happens behind the curtain is that the state has to be transfered from one page to another. This is usally done with cookies or sessions. In each request the corresponding cookie which indicates the login state is send to the server and for each request the script has to examine this cookie and check if the user is logged in.

The fact of simple dynamically typed scripting languages also plays a part in contributing to the rapid development of web applcations.

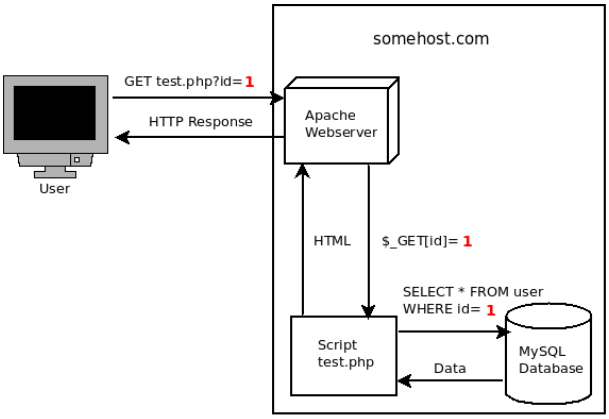

Fig. 2. SQL Injection

```
"SELECT * FROM users WHERE id = 1"
"SELECT * FROM users WHERE id = 1 OR 1=1"
"SELECT * FROM users WHERE id = 1;
  DROP TABLE users"
```

Fig. 3. Possible Vulnerable Request

It has become very easy for everybody to set up a webservice. "Introductory books [..] provide inexperienced programmers with the knowledge they need to set up [..] web applications [..] without understanding of security issues" (5)

IV. Vulnerabilities

Vulnerabilities are parts in software which can be exploited to force the software to behave in an unexpected way. In the opposite of traditional bugs these unexpected behavior typically leads to broken security requirements for example reading sensitive data or acting with special privileges.

Even if classical Buffer Overflows are no longer possible due to type saveness in modern languages used for web development, the same reasons responsible for buffer overflows now causing problems in modern application.

According to the OWASP Project(6), the two most exploited vulnerabilities in web applications are injection attacks and cross site scripting(XSS).

The technical reasons for these vulnerabilities are no or bad validations. Like traditional buffer overflow vulnerabilities injection and XSS are well known vulnerabilities and common exploits are used over and over again.

Most vulnerabilities are caused by improper validations or sanitization. While validation usually rejects input if it is not expected sanitization tries to make modifications to the input to be valid. No matter if validation or sanitization, it has to be made on every input and output. Even if validations exists there is no warranty because good validations requires a lot of effort and is a topic for its own. More about input validation can be found in (4).

A. SQL Injection

The great coding flaw to open SQL-Injection vulnerabilities is to use parameters of the requests directly for generating database queries without validation.

HTTP requests can have several parameters that are examined by the server side script. In normal GET requests these parameters are visible for the user in the URL. The most simple example is a parameter ID which indicates what data is requested. As shown in Figure 2 without validation this value flows to the database and may be directly used for constructing the query.

Not only GET but also POST parameters or even Cookies might be used for injecting code. The fact is that a variable which may contain dangerous input coming from outside is accidentally interpreted as a query command.

If an attacker finds such a vulnerability, he can modify the normal query by changing the value of ID to a tautology to read sensitive data or even try to inject code to destroy data (Figure 3).

An open SQL-Injection vulnerability can lead to one or more nasty consequences:

- Inject Malicous Data
- Privilege Escalation
- XSS-Attack
- Read sensitive Data
- Destroy Data
- Compromise the whole System
- DoS
- ...

B. Cross Site Scripting

Another great vulnerability is Cross Site Scripting or XSS for short. In this flaw it is the other way round. XSS vulnerabilities occur if output is not proper validated. The basic idea about XSS is that an attacker tries to get control over applications output. This might be based on an injection vulnerability that allows to manipulate output which is not validated.

It is always HTML which reaches the clients browser and because HTML is married with JavaScript it is common practise for browsers to trust scripts which are comming from the requested server.

But a compromised server is not aware of nasty things he is delivering. With injecting JavaScript code to a server response an attacker gets full control over clients browser. A very common exploit is to send all cookies from a client to another malicious host which collects the cookies for the attacker. Figure 4 tries to illustrate this scenario. By collecting cookies an attacker can do so called session hijacking by using another ones cookies to pretend to the applications server that he is another guy. Due to the fact that HTTP is stateless, the web application trusts the one that has the needed cookies. This attack is not targeted to the server but to the users. Operators of the applications might not even notice that there is an open XSS vulnerability until users complain that their bank account is empty.

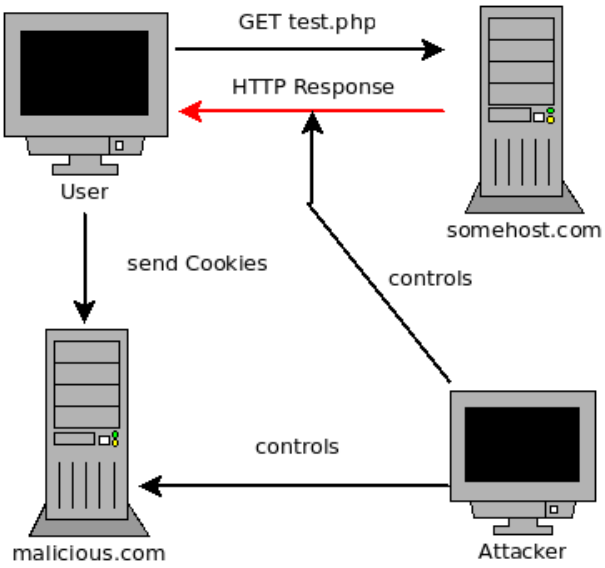

Fig. 4. Cross Site Scripting

Nasty things which can occur while having an open XSS vulnerability are as follows:

- Privilege Escalation
- Read sensitive Data
- Session Hijacking
- ...

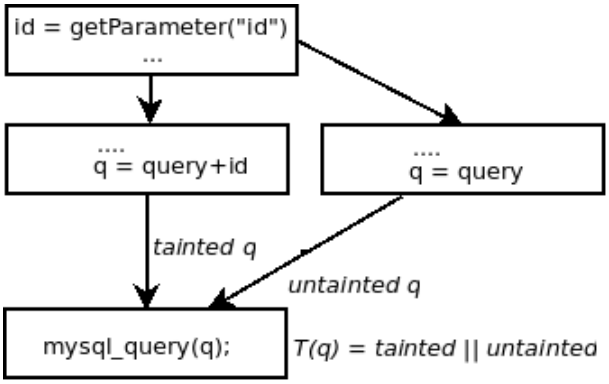

Fig. 5. Flowgraph

V. Taint Propagation

A static analysis called Taint propagation can help to find the previously described vulnerabilities within the code.

These vulnerabilities have in common that they raise from improper input or output validation.

Taint propagation is first of all a flow problem. Considering Figure 5, the control flow splits into two paths. One path where the tainted variable id, which comes from the source getParameter, is appended to q and one where q only consists of the value of query. Assuming that query is a constant and therefore not tainted at the merge point of the control flow q can either be tainted or untainted. There is a path where tainted input can reach the sensitive function mysql_query and an analysis would throw a warning for that line.

I found some approaches for Taint Propagation the interested reader may be referred to

- Taint Propgation for Java (3)
- WebSSARI for PHP (7)

First one describes an approach for web applications built with Java. They integrated their work directly into Eclipse as a plugin. Advantages of their work are an improved object naming scheme to reduce false positives and the use of Program Query Language (PQL) and Binary Decision Diagrams(BDD's).

Second is WebSSARI project which is meant to analyse PHP code. The important parts of their approach are that they handle a dynamic typed language and therefore made their analysis type aware. Their analyser is also able to insert validations automatically after it found one. This seems to be good to save time not only knowing where insecure codeparts are but also make them secure at one step.

The approaches have in common that they extended all variables by a tainteness state which is either tainted or untainted. The idea is to analyse the flow of user input and check if it can reach sensitive functions.

Additionally, descriptors are needed to determine where these interesting parts are. V.B.Libhits(3) et.al. called their descriptors like this.

- source descriptors
- derivator descriptors
- sink descriptors

Source descriptors are used to determine points where dangerous user input can come in. through code and at last sink descriptors are needed to be aware of sensitive functions which do not allow tainted input.

V.B.Libhits(3) et.al. described their descriptors formaly as:

- **Source Descriptors** $\langle m, n, p \rangle$
- **Derivation Descriptors** $\langle m, n_s, p_s, n_d, p_d \rangle$
- **Sink Descriptors** $\langle m, n, p \rangle$

We have m beeing the name of the method and n the number of parameters for this method. This is done to distinguish between methods with the same name. Java as a modern object-orientied programming language has the feature of overloading. Which means, that we can have methods with the same name but different number of parameters. Additionally, small p is the access path they definded "[...] as a sequence of field accesses, array index operations or method calls seperated by dots"(3). The notation is equal to normal programmers use every day to address fields or methods in object-orientied programming. The access path is needed escpecially for containers. Java with its great API makes much use of containerclasses like Map, Hash, Vector and so on. Because not the container object itself but the data inside such a container is dangerous the introduces access path is needed. Derivation Descriptors are a little bit more complex. They are describing "[...] how data propagates between objects in the program. They consists of a derivation method *m*, a source object given by argument number n_s, and access path p_s, and a destination object given by argument number n_d and access path p_d. This derivation descriptor specifies that at a call to method *m*, the object obtained by applying p_d to argument n_d is derived from the object obtained by applying p_s to argument n_s" (3).

To have some example descriptors, I considered their technical report(8). They collected all important descriptors by looking at Java API doc and runtime approaches which instrumented the code to find missing descriptors. Source descriptors have usually -1 as parameter number indicating that for this method no parameter is needed. Most of the time sources are getter methods which have no parameter.

Considering the second example for source descriptors this is a good example for using the introduces access paths. The returned object of *getParameterMap* is a container object whose dangerous data can only be reached with the access path *keySet().iterator().next()*.

- **Source Descriptors**
  - $\langle getParameter, -1, \epsilon \rangle$
  - $\langle getParameterMap, -1, keySet().iterator().next() \rangle$
- **Derivation Descriptors**
  - $\langle append, 1, \epsilon, -1, \epsilon \rangle$
- **Sink Descriptors**
  - $\langle executeQuery, 1, \epsilon \rangle$

Without Derivation Descriptors, the problem would be reduced to looking if a source object is used at a sink. The

```
class Vector {
  Object[] table = new Object[1024];

  void add(Object value){
    int i    = ...;
    // optional resizing ...
    table[i] = value;
  }

  Object getFirst(){
    Object value = table[0];
    return value;
  }
}
String s1 = "...";
Vector v1 = new Vector();
v1.add(s1);
Vector v2 = new Vector();
String s2 = v2.getFirst();
```

Fig. 6. Improved Object-Naming-Scheme

main reason why V.B.Libhits(3) et.al. introduced derivations are String routines of Java. "[...] Strings are immutable Java objects, string manipulation routines such as concatenation create brand new String object, whose contents are based on the original String objects"(3). Without Derivation Descriptors the analysis would find potential vulnerabilities but it might also miss some important ones.

Java is an object orientied highlevel programming language which is compiled input bytecode which is executed with a virtual machine. Their analysis do not analyse the program code itself but the generated bytecode. On the one hand it is faster but the main reason for this is that most projects rely on external libraries which might only be available in bytecode. To follow tainted data completely they also consider all libraries used by a project.

That is how they found vulnerable parts not only in the analysed projects itself but also in commonly used libraries, like for example, hibernate. "[They] have discovered an attack vector in code pertaining to the search functionality in hibernate, version 2.1.4. The implementation of method Session.find retrieves objects from a hibernate database by passing its input String argument through a sequence of calls to a SQL execute statement."(3)

A. Reduce false positives

A good context sensitive pointer analysis is needed at first because otherwise the analysis might miss important parts and would not be sound. On the other hand the false positive rate has to be as low as possible. "Containers such as hash maps, vectors, list and others are a common source of imprecision in the original pointer analysis algorithm." (3)

That is why they tried to improve the precision with an improved object-naming-scheme. As shown in 6 there is a

```
$i = (int) $_POST['index'];
$s = (stirng) $i;
echo "<hidden name=mid value='$s'>";
```

Fig. 7. Type Awareness

Vector class wich has an table array. Both Objects v1 and v2 instantiate this Vector and the analysis would consider, that v1 and v2 share that array. A tainted state might flow to s2 altough it is a completly other object which has his own vector array.

To avoid this, they associated the instancename of the object with the classname so the analysis can distinguish what container attributes belong to which instancename.

B. Type awareness

As there are many possible languages for web development today, we also have to deal with dynamic typed languages. Most used language out there might be PHP. Type of variables in PHP are only determined at runtime by value which is assigned. WEBSSARI Project (7) therefore extended PHP with additional type qualifiers. The idea is similar to "[...] the C type qualifier *const* [that] expresses the constraint that the variable can be initialized but not updated"(7). If a programmer tries to assign a variable initialized with const the compiler would notice that and throw a warning. Such additional type qualifiers are some kind of annotations. Other static analysis approaches make heavy use of annotations to have addional knowledge. But the main drawbacks of annotations are the addional effort for users to annotate their programs. In large projects this approach is not practicable. That is why WEBSSARI tried to use their type qualifiers only internally. They make use of prelude files containing method definitions for source and sink functions with extended type qualifiers.

Casting is a very common practise for sanitize user input. As shown in figure 7, which is taken from (7), there is an index parameter comming in as user input. It is casted to an integer and therefore no longer dangerous. They extended the simple tainted and untainted state with the type to reduce these kinds of false positives. The tainteness states would be tainted-string, tainted-integer, untainted-string and so on. The sensitive functions only throw a warning if tainted-strings are comming in and tainted-integers are allowed because they are not that dangerous.

C. Limits of Taint Propagation

Taint Propagation can be used very well to find missing validations of user input. But of course there are other types of vulnerabilities like

- Bruteforce
- TocTou
- DoS
- Social Engineering
- ...

which this method is not able to find.

To come up with a whole definition of sources, sinks and derivators for a programming language requires a lot of effort too. (3) Therefore used the Java-API-Doc and also did some instrumentation to find all needed descriptors.

The precision of the analysis ends where reflections or dynamic executing like PHP eval-function begins. It is hardly possible to determine what will be executes at runtime. This counts also for dynamic variables in PHP, where a simple String can be the name of a variable. Only approximation helps which might lead to a higher false positive rate.

VI. Conclusion

Taint propagation can be a quite well method for finding common vulnerabilities which are caused by improper validations. From my point of view it is a good theoretical approach but there are too few tools. In my research I found the approach of Livshits, V. Benjamin and Lam, Monica S. for analysing Java Web Applications and WEBSSARI for PHP which seems to be commercial now.

This becomes even worse due to the fact that most programmers out there do not have a right understanding of security flaws while coding.

There have to be more stable and free tools for commonly used languages such as PHP, ASP, Perl and my favorite one Ruby for analysing code. These tools have to be used so naturally like automated refacorings or version control systems.

Additionally the approach of A. Nguyen-Tuong et. al. (5) is very interesting. They tried to extend the existing PHP Interpreter with taint propagation for protect scripts at runtime. It seems this could be a fast and good working alternative but it does not open the eyes of developers to security issues itself.

For my favorite web application framework Ruby on Rails there seems to be no tool for finding security flaws. Even though there is an approach by Jong-hoon et.al. (9) for static typing of Rails. But they only tried to find bugs like null references which typically only occur at runtime.

In the end developers, are responsible for the level of trust users have in todays software. They have to take it seriously to provide the easy and most promoted but hard to achieve requirement: security.

References

[1] G. McGraw, *Software Security - Building Security In*. Software Security Series, 2006.

[2] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, 2002.

[3] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2005, pp. 18–18.

[4] B. Chess and J. West, *Secure Programming with Static Analysis*. Software Security Series, 2007.

[5] A. N.-T. et.al., "Automatically hardening web applications using precise tainting." in *In Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[6] O. W. A. S. Project, "The ten most critical web application security vulnerabilities." *http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project* 2010.

[7] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM, 2004, pp. 40–52.

[8] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis. technical report." Stanford University, 2005.

[9] J. hoon (david An, A. Chaudhuri, and J. S. Foster, "Static typing for ruby on rails."

Automated May- and Must-Analysis of Source Code

Alexander Galkin

Institute for Software Technology,
Technical University of Hamburg-Harburg
Schwarzenbergstrasse 95, 21073 Hamburg
alexander.galkin@tu-harburg.de

Abstract — *Two pillars of software validation are formal software verification (termed here may-analysis) and software testing (termed must-analysis). While formal verification requires elaborated specifications and tends to overestimate potential failures in code, good testing is based upon a duly prepared test suite and ensures that the output matches the expectations for given input so that it remains an underestimation of software quality. Both methods are implemented in software analysis tools widely used in industrial practice.*

In this manuscript, we review tools for software analysis and validation, which provide implementations of these two techniques: either just one of them or both simultaneously.

Mono-methodological tools like DART and SLAM have been in use for over a decade and were shown to be robust and reliable for software validation. They gave rise to certain commercial implementations. Combined tools like SYNERGY and DASH take advantages of both techniques and use them alternatingly to cope with those code fragments that are known to be problematic for one type of analysis and amenable to the other.

Further on, tools like SMART and SMASH use so-called compositional approach for interprocedural analysis by compiling the must- (in case of SMART) or both may- and must-summaries and make use of them to tackle the dramatically increasing complexity of software validation in real world software systems.

We attempt to provide examples of real world applications for some of the above mentioned tools with the purpose to demonstrate that they are mature enough to satisfy modern demands for software validation.

Index Terms—*software analysis, software testing, automated tests, test generation, compositional testing tools, concolic software testing, DART, SMART, SMASH, DASH, SYNERGY, SLAM*

1. Introduction

Ensuring that software meets the expectation of its users (*software validation*) is one of the most challenging tasks in software engineering. The common approach to this problem in industrial practice is to perform *software testing*, e.g. to repetitively execute the software with the input mimicking the expected user behaviour and to make sure that the outcomes of these executions satisfy the expectations.

Testing is solid in respect to the cases covered in the test suite, but can be viewed as an underestimation of software validity for it checks the software only for the test inputs from the test suite. There is always a chance that the software exhibits undesired behaviour with those inputs that were not considered during the tests.

That is why the "holy grail of software validation" [1] is (formal) *verification* which is a formal proof that the software "under test" meets its expectations for all theoretically possible test inputs and executions. In this respect, formal verification serves as an overestimation of software validity, representing the upper constraint for possible failure.

One must be, however, aware of the fact that software verification is not a magic wand, which can take the source code and judge if it is valid or not. Very much like software testing requires us first to put together an elaborated test suite, software verification necessitates careful planning while compiling thorough software specifications. These specifications are usually expressed as sets of rules or contracts, which may not be violated along any possible execution path. Complete specifications of complex software could be as complex as the software itself [1].

Both approaches have their limitations. Dijkstra's famous criticism that "[software] testing can be used to show the presence of bugs, but never to show their absence"[2] just makes the notion of *under*estimation even clearer. *Over*estimation, in turn, tends to have many false positive alarms when the detected contract violation is indeed not possible. This is due to relative imperfection of symbolic execution the formal verification heavily relies on in comparison to concrete (non-symbolic) execution testing makes use of. The former also exhibits drastic complexity grow depending upon the size of contract set and source code corpus. That is why software validation rapidly goes beyond the scalability limits of modern verification software.

The drawbacks of either method can be overcome by a wise combination of both approaches. In this case, formal verification (overestimation), combined with testing (underestimation), enables us to perform the software validation at the desirable level of precision.

In the rest of the manuscript, we will refer to the methods that exploit software testing as *must-analysis* and to the methods relying on formal verification as *may-analysis,* which represent the established terminology in this area.

Throughout, it focuses on the automated test generation and formal verification tools developed by Microsoft Research and affiliated scientists. In Section 2 we begin with a discussion about the tools that implement only one from both approaches mentioned above. Section 3 deals with their combination in so-called hybrid tools. In Section 4, we discuss some approaches that go beyond the mere combination of both methods and thereby can tackle the complexity and scalability of software validation. Section 5 then presents the conclusions about these tools and in the last section (Section 6) we speculate about further development of software validation

tools. The overview of the tools discussed here can be found in Table 1 and on Fig. 1.

TABLE I
DISCUSSED SOFTWARE TOOLS

Acronym	Summary of implemented approach(es)	Reference
SLAM	Static analysis using counter-example driven refinement	[3]
DART	Runtime testing and automated test generation	[4]
SYNERGY	Intraprocedural static analysis combining SLAM and DART	[6]
DASH	Interprocedural static analysis, improved SYNERGY	[8]
SMART	Summary-based automated test generation analysis	[4]
SMASH	Summary-based highly-scalable interprocedural static analysis, combination of SMART and DASH	[9]

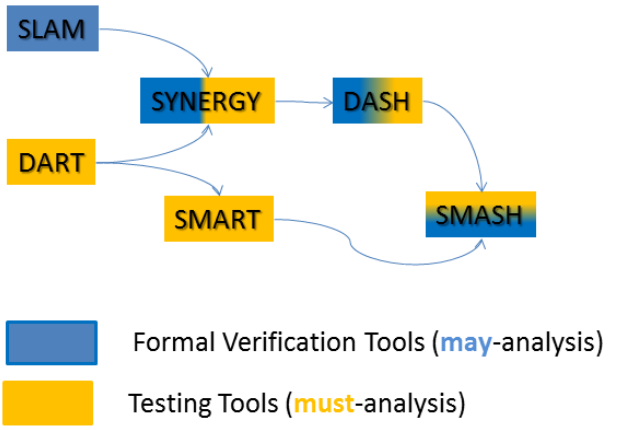

Fig. 1. Genealogy graph of software validation tools.

2. MONO-METHODICAL APPROACHES

Mono-methodical approaches exploit just one of the possible ways to validate the software and implement either formal verification algorithms (SLAM) or automated test generations (DART).

2.1. SLAM

SLAM was one of the first tools developed for automated software analysis that was implemented to increase the robustness of the third-party driver code in Microsoft Windows.

Drivers do not have direct access to the kernel code and can communicate with the kernel by calling the API functions the latter exhibits. It is then possible to derive the specification based upon the API calls to automatically validate the driver code.

The specifications are implemented using the Specification Language for Interface Checking (SLIC). A SLIC file contains the definitions of precondition contracts for certain API calls expressed in the form of short void functions containing only the variables of Boolean and enumerable types. The assertions are proved using C-like "**if**" operator and the violation is signalled using "**error**" statement, which can be likened to an "**assure (false)**" statement in C++. Depending upon the driver types these contracts are grouped into rule sets, so that the driver can be automatically checked against every rule in the set.

The SLIC code does not contain any driver code and is based solely upon the exhibited API functions. Therefore, the driver code must be first instrumented so that the SLIC specifications are integrated into its code at the sites of the API calls. This instrumentation simply adds the invocation of the SLIC pseudo-functions implementing contracts just before the actual API method call.

The instrumented program is then analyzed in the loop manner using the so-called Counter-Example Guided Refinement (CEGAR) (see Fig. 2).

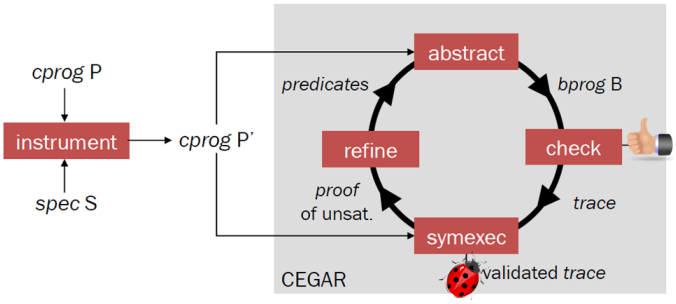

Fig. 2. Counter-Example Guided Refinement in SLAM (from [3], modified)

CEGAR begins with ultimate predicate abstraction, so that the whole source code variables are reduced to the predicate variable type. The abstracted program retains its original control structures and conditional control statements, but has a much smaller state space. Initially, it contains the minimum number of predicates, so that in the majority of conditions the predicate "**{*}**" is used. This predicate does not have a fixed Boolean value and does not depend upon any other predicates and thus represents the case when both true and false values are possible.

In this code SLAM tries to find the control flow that may eventually lead to a rule violation. The algorithm uses exhaustive exploration of the program state space in order to determine whether the error state is reachable from the program entry point. The reduction of the original program to a so-called Boolean program makes possible the exploration of the complete state space.

If there is no error state that is reachable from the entry point one can conclude that the program will not generate the appointed failure during its execution irrespective of its input (the "may"-result).

If certain error states are proved to be reachable, it does not automatically mean that the program should contain an error: reducing the program state space by predicate abstraction introduces some spurious connections between the program states, so it can well be that the error state is only reachable in

its Boolean abstraction. By the way, this is why "may" analysis is prone to false positive results.

In order to cope with the false positive alarms, SLAM tries to find the trace which leads to the assumed error state and symbolically executes the non-abstracted code in order to make sure that the state is also reachable in the original code. If the alarm was a false positive, SLAM tries to refine the original abstraction by introducing more predicates and refining the original Boolean program. The unspecific "{*}" predicate is then substituted by more specific value-dependent predicates (for instance {x>0}) that increase the precision of analysis.

The refined program is then again searched through for error states and the loop (Fig. 2) is repeated until there is either no reachable error state anymore at the given level of abstraction (positive outcome) or a trace is found leading to the error state (counterexample outcome).

The reported counterexample can be directly used to reproduce the problem in the original code. As soon as this bug is corrected the SLAM analysis can be repeated again and again until positive outcome is reached.

2.2. DART

DART stands for Directed Automatic Random Testing and is a tool for automated unit test generations in a "white-box" environment [4].

DART was the first tool that featured the so-called "concolic" approach to software testing. Concolic is a portmanteau of "concrete" and "symbolic" and denotes the testing technique where symbolic execution is used in conjunction with an automated theorem prover or constraint solver in order to generate new concrete test inputs.

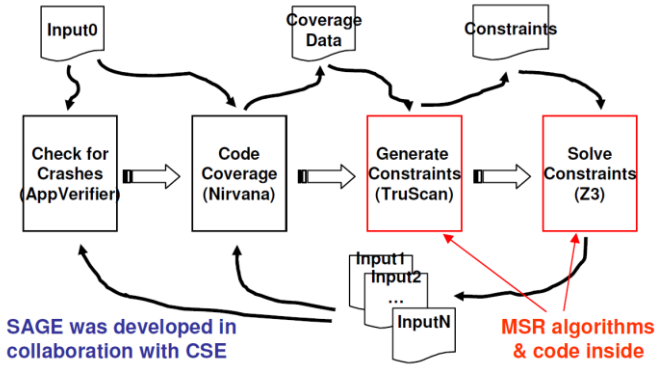

Fig. 3. General schema of concolic software testing.
(The schema illustrates the workflow of SAGE:
automated white-box testing tool developed by Microsoft Research
(MSR) together with Center for Software Excellence (CSE))

In its original implementation for C programs, DART features static analysis of the source code in order to extract the interfaces (procedure signatures, return types, etc.) further used in the test generators to perform automated testing.

The extracted interfaces are used to automatically build a test case generator that starts with purely random unit testing of the system under test. In order to keep track of the test procedure, DART relies on the code coverage metric, which is collected during the actual code execution.

Analyzing the execution paths, DART tries to guide code execution through all possible control-flow ramifications striving at maximal path coverage. As soon as all possible execution paths are covered by the generated test suite the analysis is deemed complete.

The code coverage metrics serve here as criteria for directed test generations and the predicates in control-flow ramification constructs (if, switch, ternary operator) are analyzed symbolically in order to find appropriate test cases.

DART was implemented as strictly intraprocedural analysis completely ignoring external procedure calls and assuming that all variables are properly initialized.

Besides these limitations, DART also has significant problems with scalability, because the systematic execution of all feasible program paths does not scale up to large, real-world applications. However, these drawbacks are specifically addressed in improved implementations (see Section 4.1 for more details).

2.3. Reviewing remarks

Even though the presented tools implement only a single validation algorithm, they were shown to be powerful enough to perform real-world software validation.

SLAM constitutes the core of the Microsoft Static Driver Verification tool (SDV) delivered as part of Microsoft Driver Development Kit and used to automatically check the drivers against the set of specifications (see Section 3 in [3]). This tool has been used for already a decade for formal verification of sensitive software parts – the third-party drivers – which are about to be included into Microsoft Windows Driver Repository.

The algorithm of DART has inspired many commercial software tools. One of them is the automated test generation extension for Visual Studio called PeX, which became available as a commercial tool in April 2010. PeX is tightly integrated with Visual Studio (both 2008 and 2010 are supported) and, depending upon its version, provides dynamic and static analysis features for automated test case generation using white-box testing [5].

These two examples clearly demonstrate that even mono-methodological approaches can result in robust and mature software validation tools suitable for real-world validation tasks.

3. Combined Approaches

Both software verification and testing were shown to have problems with certain source code patterns. The SLAM-like algorithms can cope well with long passages of code with many conditions and ramifications but they give up trying to analyze loops with pre- and postconditions (*while*, *do-until*).

For example look at the code snippet in Fig. 4. It depicts a while loop that is executed 1000 times before the execution bumps into "**assume()**" statement. The number of loop repetitions does not influence the potential failure in the code, which is solely dependent upon the input variable "**a**". At the

same time, these loop repetitions lead to a dramatic increase of the program state space, forcing SLAM to generate a predicate for every loop pass, what takes much time and does not help to detect the error.

```
      void foo(int a)
      {
        int i, c;
0:      i = 0;
1:      c = 0;
2:      while (i < 1000) {
3:        c = c + i;
4:        i = i + 1;
        }
5:      assume(a <= 0);
6:      error();
      }
```

Fig. 4. Code sample that is difficult for SLAM (from [6])

This code is, however, easily traversable by testing algorithms like DART, which just ignore the initial loop condition, because they do not depend upon any user input and drive the tests along two different possible paths, distinguished by the predicate in assume statement, thus revealing the code failure already during the second iteration.

On the other hand, there are code patterns that are "poisonous" for automated testing algorithms: most notably, multiple predicates which are difficult to evaluate symbolically so that the control flow can be directed appropriately.

These pitfalls of the respective analysis tools tempted many researchers to derive combined tools that could be able to analyze the code structure and decide which method of software validation to apply. Some of these tools are presented below.

3.1. SYNERGY

SYNERGY was the first algorithm that made use of both approaches and was directly implemented as a tool called YOGI [7]. SYNERGY is able to search simultaneously for bugs and proofs by combining the approaches and their results.

The algorithm of SYNERGY uses the proof under construction to guide testing. The proof construction is based upon past program executions, which produce successively more and precise *under*approximations of the error reachability tree, whereas the partition refinement (similar to predicate abstraction refinement in SLAM) is in charge of the *over*approximation precision. The partition refinement in this case relies to a great extent on the testing results that is particularly effective during long deterministic stretches of program execution, such as *for* or *while* loops [6]. Compared to the solely proof-based tools, which are also able to perform their refinement steps on loop iterations, SYNERGY takes an inexpensive, concrete approach and directly executes the code fragment containing the loop that leads to immediate refinement suggestions.

SYNERGY can also be viewed as form of property-directed testing. That means, it automatically directs its search to the program parts that are supposed to contain errors and disregards parts already proved safe.

Two major limitations of SYNERGY are intraprocedural-only code analysis and the restriction of the data flow analysis to integer variables.

3.2. DASH

DASH is the further development of SYNERGY, which is able to deal with pointer aliasing and external procedure calls.

Compared to its predecessor SYNERGY, this algorithm is capable to simultaneously maintain both a forest of test runs and a region-graph abstraction of the source code. Tests are used to locate bugs, and abstractions are used to prove their absence. On every iteration, a concrete test case is used to guide the execution toward the error region. If the error region is reached, then a bug has been found. If no more paths exist from the initial region to the error state, a proof for correctness has been found. If neither is true, then we have an abstract counterexample, which must be either confirmed by a better selected test case or waived by refining the abstraction.

DASH frequently uses the notion of a "frontier", which is the boundary between the region under test and the already tested one along an abstract counterexample that was reached by a concrete test. During every iteration, DASH tries to extend the frontier using a DART-similar automated directed test cases generator. If these tests do not reveal a concrete bug, the algorithm then tries to refine the abstract region so that the abstract counterexample is eliminated [8].

4. SUMMARY-EMPOWERED ANALYSIS

Software tools tackle the tremendous complexity of real world test systems (like operating systems) by compiling so-called "summaries" for every code not directly placed within the main program code. These are usually method invocations either from the same source code base (interprocedural analysis) or from third-party libraries. The summaries represent analysis findings at procedure boundaries and enable their smart reuse within another calling context, thereby saving the computational costs of repeated execution of this analysis.

Use of analysis summaries is widely termed as "compositional analysis".

4.1. SMART

The original algorithm of DART was further improved in the tool developed by Patrice Godefroid under the name SMART, Systematic Modular Automated Random Testing [4]. It extends the existing DART algorithms adding the abilities to test the functions in isolation and generating test summaries in form of input preconditions and output postconditions. These summaries are then reused while testing higher-level functions so that the functions are not re-evaluated. Given a bound on the maximum number of feasible paths in individual program functions, the number of explored executions by SMART thus remains linear, whereas DART exhibits the exponential grow in this bound.

SMART can be seen as the first step towards the compositional approach in software analysis, when the summaries are generated still using only a mono-methodological approach.

4.2. SMASH

SMASH (SMART + DASH) further improves the compositional approach already used in SMART by extending it to both may- and must-analysis. The summaries from may- and must-analysis are calculated on every procedure boundary and stored there. They are computed in a demand-driven manner and may use the summaries of the opposite analysis type to accelerate the computation [9].

For effective consumption of summaries the method inverts the results of the may-analysis and stores so-called "not-may" summaries. Those can then be easily used by testing algorithms to constrain the state space of the input variables.

Thus, the main advantage of SMASH is that both types of summaries are used by both types of analysis by choosing the information that could be useful to speed-up code analysis. Even though the approach is not completely new, a dexterous implementation of summary storing and retrieval on the level of algorithm allows SMASH to outperform the well-known tools like SLAM or DART.

4.3. Reviewing remarks

SMASH is now implemented as the main algorithm behind YOGI [7]. The latter is a software validation tool which will substitute SLAM in SDV in the near future.

This fact confirms just the general trends that modern software validation tools broaden their arsenal of analysis techniques by exploiting several concurrent approaches to effectively deal with real-world applications and tasks.

5. CONCLUSIONS

In this manuscript, we presented several algorithms and their existing implementations to perform software validation. These tools target the problem of automated software validation (so-called "button-click validation") in areas sensitive to the quality of code. Prominent examples are device drivers, which have direct access to the kernel API and may lead a complete system malfunction in case of software failure.

The change of the PeX testing tool status from Microsoft Research "experimental tool" to a full-fledged commercial application is just one more proof of the actuality and high industry demands for modern, highly-scalable and flexible software validation tools.

6. OUTLOOK

In the manuscript we discussed methods that allow us to perform software validation using the combinations of "may" and "must" approach. These methods, however, are only applicable given there is a complete specification we can perform our validation against. Instead of requiring the existence of such specification one might use some tools to derive specifications automatically from existing code based upon more generalized considerations.

One of such considerations is that software security and the specification to be derived must prevent the information flow vulnerability, so that no classified information is leaked to the publicly available parts of the program. That type of analysis, called "taint analysis", is common for web applications and requires no explicit specifications – those can be then automatically derived from the existing control flow by terming the existing code constructs to be "sources" (producing the taint), "ordinary nodes" (forwarding the information without affecting the taint, e.g., propagating the taint if it is present in the input), "sanitizers" (freeing the input information from the taint), and "sinks" (structures that present information publicly, should never get any part of taint). Taint analysis can help to automatically derive specification information [10], which can be used as a part of the final specifications.

ACKNOWLEDGMENT

I would like to thank Prof. Sibylle Schupp for topic suggestion and her helpful and detailed feedback during work on this manuscript, especially what regards her very valuable comments regarding the correct use of English language.

REFERENCES

[1] Beckman, N.E., Nori, A.V., Rajamani, S.K, Simmons, R.J., Tetali S.P. and Thakur A.B.. *Proofs from Tests,* IEEE Transactions on Software Engineering (RapidPosts), vol. 99, 2010.

[2] Dijkstra, E. W. *The humble programmer*, Commun. ACM 15, vol. 10, pp. 859-866, 1972.

[3] T Ball, V Levin, SK Rajamani, *A Decade of Software Model Checking with SLAM*, Microsoft Research internal publication, 2010.

[4] P. Godefroid. *Compositional dynamic test generation*. In POPL '07: In "Principles of Programming Languages", pp. 47–54, 2007.

[5] Tillmann, N. and De Halleux, J.. *Pex: white box test generation for .NET.* In "Proceedings of the 2nd international Conference on Tests and Proofs" (Prato, Italy, April 09 - 11, 2008), 2008.

[6] Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.. *SYNERGY: A new algorithm for property checking*. In "FSE 2006: Foundations of Software Engineering", ACM Press, New York, pp. 117–127, 2006.

[7] Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: *The Yogi Project: Software Property Checking via Static Analysis and Testing.* In "TACAS 2009: Tools and Algorithms for Constuction and Analysis of Systems", LNCS, Springer, Heidelberg, vol. 5509, pp. 178–181, 2009

[8] Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: *Proofs from tests.* In "ISSTA 2008: International Symposium on Software Testing and Analysis", ACM Press, New York, pp. 3–14, 2008

[9] Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: *Compositional May-Must Program Analysis: Unleashing The Power of Alternation.* Microsoft Research Technical Report MSR-TR-2009-2, Microsoft Research, 2009

[10] Livshits, B., Nori, A.V., Rajamani, S.K., Banerjee, A.. *Merlin: Specification Inference for Explicit Information Flow Problems,* In "PLDI 2009: Programming Language Design and Implementation", ACM Press, New York, 2009.

Statische Fehlererkennung

Felix H. Wenk
Universität Bremen
Bremen
Email: fwenk@informatik.uni-bremen.de

***Zusammenfassung*—In dieser Arbeit werden drei Verfahren zur statischen Fehlererkennung anhand konkreter Implementierungen betrachtet, nämlich Model-Checking, Fehlererkennung mit statischen Typsystemen und Meta-Level-Compilation. Als statische Fehlererkennung bezeichnet man die Erkennung von Fehlern in Programmcode ohne diesen, im Gegensatz zum Testen, ausführen zu müssen. Mittels Model-Checking werden anhand einer formalen Spezifikation Eigenschaften des Programms anhand eines Modells des Programms verifiziert. Bei der Fehlererkennung mit statischen Typsystemen werden Typfehler und einfache Programmierfehler erkannt. Erweitert man das Typsystem, lassen sich auch Ressourcenmanagement-Fehler erkennen. Letzteres ist verbunden mit großen Änderungen am originalen Code. Mittels Meta-Level-Compilation lassen sich mit von der Länge des zu analysierenden Programms unabhängigem Aufwand Implementierungsfehler in einem Programm finden.**

I. Einführung

Software enthält Fehler. Das heißt das tatsächliche Verhalten der Software weicht von ihrem – leider nicht immer explizit – spezifizierten Verhalten ab. Fehler sind in jedem Programm ab einer bestimmten Größe nahezu unvermeidlich. Sowohl bei der Analyse bestehender Software und ganz besonders während des Entwicklungsprozesses neuer Software sind wir sehr daran interessiert, mit möglichst geringem Aufwand möglichst viele Fehler zu finden.

Typischer Weise wird, um das tatsächliche Verhalten eines Programms mit dem gewünschten Verhalten zu vergleichen, also um Fehler zu finden, getestet. Dies geschieht mehr oder weniger formal, beispielsweise mit so genannten Unit-Tests, bei denen einzelne Softwarekomponenten mit bestimmten Eingaben ausgeführt werden. Oft besteht der Test aber auch nur darin, die Software als Ganzes einfach auszuführen. In jedem Fall ist das Testen dynamisch, das heißt, das potentiell fehlerhafte Programm wird tatsächlich ausgeführt.

Letzteres bringt eine Reihe Probleme mit sich. Es muss sichergestellt werden, dass wirklich das Verhalten der gesamten Software getestet wird, also der ganze Code ausgeführt wird. Das macht beispielsweise die Testfälle für Unit-Tests kompliziert und deren Anzahl erföht sich im gleichen Maße, in dem das zu testende Prgramm wächst. Zusätzlich bestehen bei vielen Komponenten innerhalb eines Programms Abhängigkeiten, so dass diese gar nicht für sich testbar sind und ein Test andere Komponenten beeinflussen würde.

Schlimmstenfalls lässt sich das Programm, in dem Fehler gefunden werden sollen, gar nicht testen. Ein typisches Beispiel daführ sind Gerätetreiber in Betriebsystemen, für die konkrete Hardware vorhanden sein muss, um sie zu testen. Außerdem kann ein positiver Test, also das Finden eines Fehlers zur Laufzeit, in manchen Situationen ernsthaften Schaden verursachen. Die Folgen können dramatisch sein, wenn zur Laufzeit Fehler in der Steuerungssoftware eines Roboters gefunden werden.

Die Alternative zum Testen ist die statische Fehlererkennung. Dabei wird versucht, Fehler im Programmcode zu finden, ohne ihn auszuführen. Dies kann man natürlich, indem man den vorhandenen Programmcode aufmerksam durchliest. Bei großen Programm ist das aber wahrscheinlich wenig effektiv. In den folgenden drei Abschnitten werden drei verschiedene Ansätze betrachtet mit denen man auch in großen Programmen statisch Fehler finden kann.

In Abschnitt 2 betrachten wir das Model Checking. Am Beispiel der beiden Tools Uppaal [1] und Bandera [2] wird gezeigt, wie sich in einem Programm mittels eines Modells und einer Spezifikation des gewünschten Verhaltens Fehler finden lassen.

In Abschnitt 3 werden ein statisches Typsystem und die Typsystemerweiterung Vault [3] zur statischen Fehlererkennung vorgestellt, bevor in Abschnitt 4 die Meta-Level Compilation [5] zur statischen Fehlererkennung beschrieben wird. Anhand einer Erweiterung [4] der Meta-Level Compilation wird im zweiten Teil von Abschnitt 4 gezeigt, wie sich automatisch Annahmen über das gewünschte Verhalten generieren und auch überprüfen lassen.

Zum Schluss werden die Ergebnisse der Abschnitte 2 bis 4 zusammengefasst und kurz bewertet.

II. Model Checking

Eine Variante der statischen Fehlererkennung ist die Verifikation mittels Model-Checking. Zum Model-Checking gehören im Wesentlichen drei Komponenten: ein Modell, eine Spezifikation der Eigenschaften, die am Modell überprüft werden sollen und ein Verfahren, um die eigentliche Überprüfung durchzuführen. Die drei Komponenten werden im Folgenden, die ersten beiden am Beispiel des Tools Uppaal, [1] betrachtet.

A. Verifikation mit Timed Automata

Uppaal verwendet als Modell zur Verifikation ein Netzwerk aus so genannten Timed Automata [1]. Vereinfacht kann man sich einen Timed Automaton als einen endlichen Automaten vorstellen, zu dem Clock-Variablen, ganzzahlige Variablen, Guards, Invarianten und Aktionen hinzukommen. Variablen können globale oder lokale Gültigkeit haben. Außerdem können Automaten parametrisiert werden. Abbildung 1 zeigt

einen Timed Automaton wie Uppaal ihn darstellt. Ein Zustand des Timed Automaton wird beschrieben durch den Control State oder Location, in der er sich befindet (blaue Kreise in Abbildung 1), die Belegung der Variablen und die gültigen Bedingungen bzgl. der Clock-Variablen. Mittels letzterer wird die kontinuierliche Zeit modelliert. Diese ist natürlich nicht explizit darstellbar, allerdings lassen sich Bedingungen wie „Clock-Variable c ist kleiner 10“ formulieren. Letztere lassen sich als Invarianten für eine Location verwenden. Solange sich der Timed Automaton in einer Location befindet, müssen deren Invarianten erfüllt sein. Sind sie es nicht, muss die Location verlassen werden. Ähnlich verhält es sich mit Guards. Guards beziehen sich auf die Kanten zwischen den Locations (grün in Abbildung 1). Eine Kante kann genommen werden, wenn alle Guards dieser Kante erfüllt sind. Wenn eine Kante genommen wird, können damit Aktionen verbunden werden (blau). In diesen lassen sich Clock-Variablen zurücksetzen oder Werte der anderen Variablen ändern. Im Startzustand befindet sich der Automat in der Start-Location (gekennzeichnet durch einen inneren Kreis) mit einer initialen Variablenbelegung und die Clock-Variablen sind initial 0.

B. Computation Tree Logic (CTL)

Am so beschriebenen Modell sollen nun Eigenschaften überprüft werden. Dazu müssen diese in einer für den Model-Checker verständlichen Form formuliert werden. Für Uppaal wird dazu eine eingeschränkte Form der Computation Tree Logic (CTL) [1] verwendet. Diese besteht im Wesentlichen aus State- und Path-Formeln.

State-Formeln sind Formeln, die sich auf einen einzelnen Zustand des Modells beziehen. Beispiele dafür sind $\phi_1 = v < 5$ für „die Variable v ist kleiner 5“ oder $\phi_2 = P(0).Waiting$ für „der Prozess 0 ist in der Location Waiting“. Zusätzlich können State-Formeln zu einer neuen State-Formel logisch verknüpft werden, zum Beispiel $\phi = \phi_1 \wedge \phi_2$.

Path-Formeln sind State-Formeln, die mit $[]$ oder $\diamond$ qualifiziert sind. Dabei bedeutet $[]\phi$, dass ϕ auf dem gesamten Pfad, also auf allen Zuständen einer Zustandsfolge erfüllt ist, wenn $[]\phi$ erfüllt ist. Bei $\diamond\phi$ reicht es, wenn ein Zustand des Pfades ϕ erfüllt.

Path-Formeln lassen sich nun über Pfade quantifizieren. $A[]\phi$ bedeutet dabei, dass $[]\phi$ auf allen Pfaden gilt, wenn $A[]\phi$ gilt und $E[]\phi$, dass es einen Pfad gibt, auf dem $[]\phi$ gilt. $A \diamond \phi$ und $E \diamond \phi$ verhalten sich analog.

C. Verfahren

In Abbildung 1 ist eine fehlerhafte Implementierung von Petersons Algorithmus abgebildet, die gegenseiten Ausschluss gewährleisten soll. Es dürfen sich also nicht mehrere Prozesse gleichzeitig in der Critical Section (cs) befinden. Dies können wir nun für Uppaal folgendermaßen formulieren: $A[]\forall i : \forall j : P(i).cs \wedge P(j).cs \rightarrow i = j$ [6]. Nach genauerer Betrachtung des Modells stellen wir fest, dass diese Formel nicht erfüllt ist. Bei zwei Prozessen, $pid = 1$ und $pid = 2$, kann es folgende Abfolge geben. 1 nimmt die erste Kante und setzt die turn-Variable auf 1. 2 nimmt alle möglichen Kanten bis er sich in

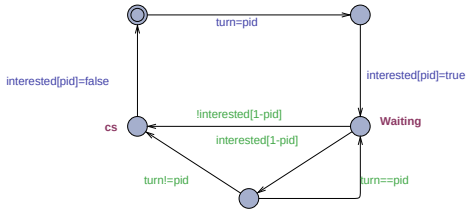

Abbildung 1. Petersons Algorithmus mit Fehler

der Location *cs* befindet. Da nun $turn = 2$ ist, kann 1 nun ebenfalls in die Location *cs*. Lässt man das Modell mit Uppaal verifizieren, wird diese Abfolge auch als Begründung dafür ausgegeben, dass die Formel nicht gilt. Aber wie funktioniert das?

Sehr stark vereinfacht kann man sich die Verifikation folgendermaßen vorstellen (nach [7]). Zuerst wird ein so genannter Compuation Tree aufgebaut, also ein Baum, in dem die Pfade von der Wurzel zu jedem Blatt alle möglichen Zustandsfolgen repräsentieren. Außerdem wird ein Syntaxbaum der zu überprüfenden Eigenschaft aufgebaut. In diesem wird nun ausgehend von den Blättern zur Wurzel jede Teilformel genommen und jeder Zustand im Computation Tree, so er die Teilformel erfüllt, damit annotiert. Nachdem man mit der Gesamtformel annotiert hat, erfüllt das Modell die Formel, wenn die Startzustände annotiert wurden. Bevor man dieses sehr vereinfachte Verfahren anwenden kann, muss die Formel eventuell in eine äquivalente Form umgeformt werden [7].

D. Bandera: Automatische Modellextraktion

Bevor man verifizieren kann, muss man für Uppaal die Abstraktion des Programms erst einmal selbst modellieren. Das ist zeitaufwändig und es lassen sich dabei beliebig viele Fehler machen, sodass es als Bestandteil des Entwicklungsprozesses in den meisten Fällen nur in sehr geringem Maße in Frage kommt.

Um dieses Problem etwas zu abzumildern, lassen sich Modelle auch automatisch aus Programmcode generieren, deren Anpassung deutlich weniger Aufwand erfordert, als das gesamte Modell per Hand zu modellieren. Wie man das machen kann, wird hier am Beispiel von Bandera [2] gezeigt.

Bandera generiert aus Programmcode in der Programmiersprache Java Eingaben für Model-Checker, beispielsweise für den Model-Checker SPIN. Der Ablauf der Übersetzung ähnelt dem Ablauf eines Compilers für Programmiersprachen. Zuerst wird der Java-Programmcode auf Jimple [2], eine Repräsentation des Programms mit der Bandera weiterarbeitet, übersetzt. Dabei bleiben die Beziehungen zwischen den Jimple-Konstrukten und dem Java-Code erhalten. Die Abbildung kann also auch umgekehrt werden. Auf dieser Jimple-Fassung des Programms werden nun verschiedenen Transformationen, namentlich Slicing und Abstraktion durchgeführt.

Für das Slicing generiert Bandera basierend auf den am Modell zu überprüfenden Eigenschaften ein Slicing Criterion [2]. Im Slicing Criterion stehen alle Statements des Programms, die die zu überprüfenden Eigenschaften beeinflussen.

Während des eigentlichen Slicings werden nun nur Programmteile entfernt, die keinerlei Einfluss auf die im Slicing Criterion enthaltenen Statements haben. Nach dem Slicing hat man also ein deutlich kleineres Modell des Programms, das weniger Teile des Programms modelliert, sich aber bezüglich der Eigenschaften, die überprüft werden sollen, wie das originale Programm verhält.

Nach dem Slicing wird nun mit Unterstützung des Benutzers die Abstraktion durchgeführt. Dabei wird von den im Programm enthaltenen Variablen und von den auf diesen Variablen angewendeten Operationen abstrahiert, um die im Modell vorhandenen Zustände zu verkleinern.

Beispielsweise könnte an einer bestimmten Stelle im Programm eine Variable vom Typ 32-Bit-Integer verwendet werden. Sowohl für den Programmfluss als auch für die Eigenschaften, die überprüft werden sollen, mag aber nur interessant sein, ob der Wert dieser Variable größer als 0 ist, oder nicht. Man kann den Integer also auf eine Variable vom Typ Boolean abstrahieren. Entsprechend müssen die Operationen auf der Variablen abstrahiert werden. Diese ändern nun nicht mehr den tatsächlichen Wert des Integers, sondern nur noch ob die Integer gößer oder kleiner bzw. gleich 0 ist. Die Variable trägt also zu unserem Zustandsraum nur noch 2 statt 2^{32} Zustände bei.

In Bandera stehen für die verschiedenen Datentypen unterschiedliche Abstraktionen in einer Abstraktionsbibliothek zur Verfügung. Zusätzlich können Abstraktionen in einer speziellen Sprache definiert werden. Wurde für einen Variable eine Abstraktion gewählt, wird jedes Vorkommen der Variable im Programm durch die Abstraktion ersetzt. Sollten sich dadurch Uneindeutigkeiten im Programmfluss ergeben, also z.B. eine Bedingung nicht mehr eindeutig ausgewertet werden können, werden alle nach dieser Bedingung möglichen Verzweigungen mit ins Modell aufgenommen.

Nach den Transformationsschritten wird die nun verkleinerte und abstrahierte Jimple-Fassung in die „Bandera Intermediate Language“ [2] übersetzt. Aus dieser können nun die verschiedenen Back-End-Komponenten Eingaben für die eigentlichen Model-Checker generieren.

Abschließend lässt sich zum Model-Checking festhalten, dass sich, vorausgesetzt das Modell und die Spezifikation der Eigenschaften sind richtig, logische Fehler in Programmen mit Model-Checkern zuverlässig finden lassen wie im Beispiel der Fehler im Verfahren für den gegenseitigen Ausschluss. Allerdings sind die Formulierung der Eigenschaften und die Generierung des Modells aufwändig und, wenn diese nicht weitgehend automatisiert ist, fehleranfällig. Außerdem besteht die Gefahr, dass Programmierfehler bei der manuellen Modellierung nicht mitmodelliert werden. Besonders in diesem Fall wirkt es sich nachteilig aus, dass eben nicht das tatsächliche Programm sondern nur ein Modell dessen verifiziert wird. Dennoch besteht die Aussicht darauf, dass sich die Verifikation mittels Model-Checking mit zunehmender Automatisierung der Modellgenerierung in den Softwareentwicklungsprozess integrieren lassen wird.

III. Typsysteme

Fehler kann man statisch aber auch viel einfacher erkennen. Die wahrscheinlich am häufigsten eingesetzte Methode ist wohl die statische Fehlererkennung mittels eines statischen Typsystems. Bei einem Programm, das in einer entsprechenden Programmiersprache geschrieben wurde oder wird, kommt diese Methode jedes Mal zum Einsatz, wenn das Programm übersetzt wird. Beispielhaft betrachten wir das statische Typsystem der Programmiersprache C [8].

A. Statisches Typsystem (C)

C kennt vier Basisdatentypen [8]. Dies sind *char*, der eine ein Byte lange ganze Zahl repräsentiert und typischer Weise verwendet wird, um ein Schriftzeichen darzustellen, *int* für längere ganze Zahlen, *float* für 32-Bit-Gleitkommazahlen und *double* für 64-Bit-Gleitkommazahlen. Diese Typen können noch verschieden qualifiziert werden (z.B. *unsigned int* für ganze Zahlen größer oder gleich 0), außerdem gibt es Zeiger und Arrays der entsprechenden Typen.

In einem C-Programm ist jede Variable oder Konstante von einem solchen Typ. Wird nun eine Operation, die mit Werten eines bestimmten Typs arbeitet, ausgeführt, kann zur Kompilierzeit überprüft werden, ob die Operation auf den entsprechenden Variablen oder mit den entsprechenden Parametern überhaupt angewendet werden kann.

Im folgenden Code findet ein C-Compiler typischer Weise Fehler:

```
void do_something(double *var);
/* Code */
int *pointer_to_integer;
/* Code */
do_something(pointer_to_integer);
```

Versucht man diesen Code mit *gcc* zu übersetzen, erhält man folgenden Hinweis:

```
typesystemdemo.c:24: warning:
    passing argument 1
    of 'do_something' from
    incompatible pointer type
typesystemdemo.c:10: note:
    expected 'double *'
    but argument is of type 'int *'
```

Der Fehler wurde also schon zur Kompilierzeit gefunden. Mittels des Typsystems lässt sich also schon zur Kompilierzeit bestimmen, welche Operationen überhaupt anwendbar sind.

Was aber passiert mit folgendem Code-Abschnitt?

```
void ressource_management_bug()
{
        int i; char *p = malloc(1337);
        /* Million lines of code */
        free(p);
        printf("Bug, bug, bug: %d\n", i);
        /* Code without malloc */
        i = p[0] + p[1] + p[2];
```

```
        /* Some lines of code. */
        free(p);
}
```

Bei diesem Code fällt dem Compiler kein Fehler zur Kompilierzeit auf. Dennoch sind viele Fehler enthalten. Ein Fehler ist Beispielsweise die doppelte Verwendung von *free()* auf demselben Zeiger. Beim zweiten Aufruf würde Speicher freigegeben werden, der dem Prozess, der das Programm ausführt, gar nicht mehr gehört. So einen Fehler kann man nicht mit Cs Typsystem finden. Schließlich ist am Aufruf *free(p)* prinzipiell nichts auszusetzen.

Um dennoch zur Kompilierzeit solche Fehler mittels des Typsystems finden zu können, muss man das Typsystem erweitern. Und zwar muss nicht nur die Information enthalten sein, welche Operationen prinzipiell anwendbar sind, sondern auch die Information welche Operationen an einer bestimmten Stelle im Programm anwendbar sind [3].

Ein Beispiel für so eine Erweiterung ist Vault [3].

B. Erweiterung des Typsystems (Vault)

Vault erweitert Cs Typsystem um *Type Guards* und *Keys*. Ein Type Guard ist eine Bedingung, die erfüllt sein muss, damit in einem Programm auf eine Ressource, also z.B. eine Variable, zugegriffen werden darf. Ein *Key* ist ein Kompilierzeit-„Token“ [3], der für eine bestimmte Laufzeit-Ressource steht. Zu den Keys gibt es das *Held-Key-Set*. In dieser Menge sind alle Keys enthalten, deren Runtime-Ressourcen an einer bestimmten Stelle des Programms zur Verfügung stehen. Hiermit lassen sich nun die einfachsten Type Guards beschreiben. Auf eine Ressource darf nun zugegriffen werden, wenn sich ihr entsprechender Key im Held-Key-Set befindet.

In folgendem Code gibt es drei Ressourcen, nämlich *j, k, l*. Jeder dieser Ressourcen ist ein Key zugeordnet. Dieser befindet sich jeweils links vom Doppelpunkt in der Variablendeklaration.

```
J:int j = 4711;
K:int k;
L:int l = 9773;
/* More code */
/* Line xy */
/* More code */
```

Nun haben wir an der Stelle *Line xy* das Held-Key-Set $\{J, L\}$. Es darf also in *Line xy* auf die Ressourcen *j* und *k* zugegriffen werden.

Um eine Beziehung zwischen einem Key und eine Objekt herzustellen, werden in Vault die so genannten *Tracked Types* verwendet.

„`tracked(K) point p = new tracked point x = 3; y = 4;`“ [3] stellt eine 1-zu-1-Beziehung zwischen dem Key K und dem Point-Objekt dar [3]. Dabei ist der Typ des Objektes der Variablen *tracked(K) point*. Ab hier bezeichnen alle Variablen vom Typ *tracked(K) point* dasselbe Objekt. Der Key *K* wird bei der Konstruktion des Objektes dem Held-Key-Set hinzugefügt. Es darf also nun auf das Objekt zugegriffen werden.

Operationen bzw. Funktionen, die Objekte verändern, gibt es in Vault natürlich auch. Diese müssen allerdings alle Änderungen, die sie am Objekt machen, im Held-Key-Set nachvollziehen. Dazu erhalten Funktionen eine Effect-Clause. Diese Beschreibt Vor- und Nachbedingungen bzgl. der Keys der Objekte, die die Funktion verändert. Beispielsweise muss für die oben schon angesprochene Funktion *free()* der Key des Objektes im Held-Key-Set sein. Wenn die Funktion zurückkehrt, wird der Key aus dem Held-Key-Set entfernt.

```
void free(tracked(K) T o) [-K]
```

Im Beispiel von *free()* wird also ein Argument vom Typ *tracked(K) T* erwartet, dessen Key im Held-Key-Set ist. Mit *[-K]* wird beschrieben, dass der Key aus dem Held-Key-Set entfernt wird. Funktionen können auch Keys zum Held-Key-Set hinzufügen oder ganz neue Keys generieren.[1]

Das Management des Held-Key-Sets wirft allerdings ein Problem auf. Wenn der Programmfluss anhand einer Bedingung verzweigt, die erst zur Laufzeit ausgewertet werden kann, in einem Zweig ein Key aus dem Held-Key-Set entfernt wird, im anderen aber nicht, kann zur Kompilierzeit nach der Zusammenführung der beiden Zweige nicht mehr entschieden werden, ob der entsprechende Key im Held-Key-Set ist, oder nicht.

Dieses Problem wird mittels Keyed Variants gelöst. Eine Keyed Variant ist ein Typ, der mit einem Key parametrisiert wird. Ähnlich wie in funktionalen Programmiersprachen wird der Typ über Konstruktoren konstruiert. Ein Konstruktor kann nun ebenfalls mit dem Key parametrisiert sein. Wird mit einem solchen Konstruktor eine Keyed Variant konstruiert, wird der Key aus dem Held-Key-Set entfernt und in der Keyed Variant „zwischengespeichert“. Mittels Pattern-Matching kann im späteren Programmverlauf abgefragt werden, ob ein Konstruktor mit Key-Parameter oder ohne genutzt wird. Im ersten Fall wird der Key dann wieder in das Held-Key-Set zurück getan, im letzteren nicht. So lassen sich auch Fehlerwerte von Funktionen modellieren. Beispielsweise könnte ein Vault-Variante von *malloc()* eine Keyed Variant zurückgeben, auf die dann gematcht werden kann. Wurde ein Konstruktor mit Key-Parameter verwendet, ist der Key nun im Held-Key-Set und die Ressource, für die mit *malloc()* Speicher reserviert wurde, kann verwendet werden.

Ein anderes Problem ist, dass zur Kompilierzeit nicht für beliebig viele Objekte, die zur Laufzeit erzeugt werden, Keys vergeben werden können. Vault behilft sich mit so genannten anonymen Keys. Diese können allgemein nicht explizit benannt werden, können aber für Funktionen wie das oben gezeigte *free()*, das zu einem Objekt den Key im Held-Key-Set erwartet, verwendet werden.

Programmcode, der mit Vault geschrieben wurde, ist erwartungsgemäß länger als entsprechender Code in C,

[1] Keys haben auch noch einen Zustand, der durch Funktionen verändert werden kann. Der Zustand von Keys wird hier allerdings nicht behandelt.

schließlich wird mit dem erweiterten Typsystem mehr kodiert. DeLine et al. brauchten für die Implementierung des Floppy-Disk-Drivers für Windows 2000 mehr als 5% mehr Code als die originale C-Variante. Die Umstellung der Interfaces in existierender Software ist eine zusätzliche Hürde, wenn Vault eingesetzt werden soll [3].

Vom Standpunkt der Analyse existierender Software, die für die Analyse mit Vault aufbereitet wurde, muss beachtet werden, dass die Analyse sich dann nicht mehr auf das originale Programm bezieht, sondern auf das Programm plus die Änderungen für Vault. Ähnlich wie bei der manuellen Modellgenerierung für das Model-Checking können hier natürlich wieder Fehler passieren. Während der Softwareentwicklung kann Vault aber gerade beim Ressourcenmanagement fatale Fehler vermeiden.

IV. Meta-Level Compilation

Ganz ohne Veränderungen am eigentlichen Programm kommt das dritte Verfahren zur statischen Fehlererkennung, die „Meta-Level-Compilation“ [5], aus. Um die dem Verfahren zugrundeliegende Idee zu verstehen, betrachte man folgendes Beispiel. Angenommen wir wollen den Linux-Kernel statisch nach Fehlern absuchen.[2] Beispielsweise könnte dabei folgender Code anzutreffen sein.

```
cli();
/* Dinge tun */
sti();
```

Die Semantik dieses Codefragments auf der Ebene des Compilers ist offensichtlich. Zuerst wird die Funktion *cli()* aufgerufen, anschließend werden Dinge getan, wonach die Funktion *sti()* aufgerufen wird. Dies ist aber eben nur die Semantik auf Compilerebene. Der Programmierer, der ein solches Code-Fragment schreibt, möchte eigentlich zuerst die Interrupts ausschalten (*cli()*). Anschließend möchte er Dinge tun, während derer keine Interrupts auftreten dürfen. Sie müssen also ausgeschaltet sein. Danach werden die Interrupts wieder eingeschaltet (*sti()*), um den normalen Betrieb des Linux-Kernels fortzusetzen. Engler et al. nennen diese Bedeutung „Meta-Level Semantik“ [5].

Bei der Meta-Level-Compilation geht es nun darum, diese Bedeutung ebenfalls zu kompilieren, um anhand derer statisch Fehler erkennen zu können.

A. metal & xgcc: Compiler-Erweiterungen

Möglich ist dies mit der Sprache Metal [5]. Mit dieser Sprache lassen sich Automaten beschreiben, die ihre Zustände dann wechseln, wenn ein bestimmtes Konstrukt im Quellcode gefunden wird.

Ein Zustand in Metal wird durch eine Menge aus Regeln beschrieben. Eine Regel ist dabei ein Tripel aus einem Pattern, einem Zielzustand und einer Aktion. Patterns werden in der Sprache verfasst, in der der zu überprüfende Code geschrieben ist. Für den Linux-Kernel wäre so ein Pattern also ein C-Konstrukt.[3]

Matcht nun ein Pattern des aktuellen Zustandes mit Programmcode, so wird in den Zielzustand gewechselt und die zugehörige Aktion ausgeführt. Der Zustandswechsel und die Ausführung der Aktion sind dabei optional. Es kann also ein Zustand gewechselt werden, ohne eine Aktion auszuführen und eine Aktion kann ausgeführt werden, ohne das der Zustand gewechselt wird.

Zur Veranschaulichung dient folgendes, vereinfachtes Beispiel aus [5], mit dem sich der Beispielcode weiter oben überprüfen ließe.

```
sm check_interrupts {
  pat enable = { sti(); };
  pat disable = { cli(); };
  is_enabled: disable ==> is_disabled
    | enable ==> { err("double enable"); };
  is_disabled: enable ==> is_enabled
    | disable ==> { err("double disable"); };
    | $end_of_path$ ==>
      { err("exisiting w/intr disabled!"); };
}
```

Hier werden zuerst die beiden Patterns *enable* und *disable* definiert, die mit den entsprechenden Sprachkonstrukten matchen. Anschließend werden die beiden Zustände *is_enabled* und *is_disabled* definiert. Matcht im Zustand *is_enabled* das Pattern *disable* wird in den Zustand *is_disabled* gewechselt. Matcht das *enable* Pattern, wird die Aktion ausgeführt, eine Fehlermeldung auszugeben, dass Interrupts angeschaltet werden, obwohl sie schon angeschaltet sind. *is_disabled* ist analog definiert. Dort wird allerdings noch das spezielle Pattern benutzt, das mit dem Ende einer Funktion matcht. Ist dies der Fall, wird eine entsprechende Fehlermeldung ausgegeben – in Linux-Kernel-Code ist es normalerweise ein Fehler, eine Funktion zu verlassen ohne Interrupts wieder eingeschaltet zu haben.

Solche Metal-Automaten können nun mit dem Metal-Compiler *mcc* [5] kompiliert werden. Dieses Kompilat wird als Eingabe für eine Erweiterung von *gcc*, *xgcc*, verwendet. *xgcc* wendet, mit dem Automaten und dem eigentlich Programm als Eingabe, nun den Automaten für jeden Pfad durch jede Funktion des Programms an.[4]

Wir haben nun zu einer Annahme über den Programmcode, nämlich das auf ein *cli()* ein *sti()* folgen muss, einen Automaten geschrieben, mittels dem die Annahme überprüft werden kann. Aber welche Annahmen kann man noch treffen? Der Programmierer, der den Programmcode geschrieben hat, wird sicherlich noch mehr Automaten schreiben können, die seine eigenen Annahmen über seinen eigenen Code kontrollieren. Allerdings fällt auf, dass viele Annahmen eine ähnliche

[2] Dies ist tatsächlich einer der Anwendungsfälle für MLC [5].

[3] Patterns können neben dem Sprachkonstrukt auch noch vorgefertigte Metal-Konstrukte enthalten, mit denen z.B. Argumente für Funktionen oder case-Labels [5] beschrieben werden können.

[4] Um nicht unendlich viele verschiedene Pfade zu generieren, werden insbesondere Schleifen speziell behandelt [5].

Form haben. Etwas abstrakter ist *cti()-sti()* das paarweise Auftreten zweier Funktionsaufrufe. Annahmen, die in solche Muster passen, lassen sich auch automatisch generieren und überprüfen [4].

B. Automatische Regel-Generierung

Engler et al. nennen solche Annahmen auch „Beliefs". [4] Wie oben schon bemerkt, passen diese Beliefs meistens in entsprechende Muster. Solche Muster oder auch „Belief-Templates" sehen etwa folgendermaßen aus: *„do not dereference null pointer* $< p >$" [4] oder *„Lock* $< l >$ *protects variable* $< v >$" [4].

Die spitzen Klammern symbolisieren Slots. $< p >$ ist ein Slot für einen Pointer. Setzt man einen konkreten Pointer für p in den Slot ein, wird dieser Pointer „Slot-Instanz" genannt. Das Belief-Template *„do not dereference null pointer* $< p >$" spezifiziert also Annahmen über alle Pointer.

Über derartige Templates spezifizierte Annahmen lassen sich sich zwei Gruppen einteilen: „Must-Beliefs" und „May-Beliefs" [4]. Findet man eine Stelle im Programm, in der ein Must-Belief verletzt wird, findet man damit immer einen Fehler. Wie man einen Must-Belief überpfüft, lässt sich an folgendem Beispiel, das Engler et al. im Linux-Kernel gefunden haben, nachvollziehen. Überprüft wird der Null-Pointer-Belief mit der Slot-Instanz, also dem Pointer, *tty*.

```
struct mxser_struct *info=tty->driver_data;
unsigned long flags;

if (!tty || !info->xmit_buf)
        return 0;
...
```

Zuerst wird für die Slot-Instanz ein Belief-Set angelegt. Im Belief-Set sind die Zustände, die die Slot-Instanz an einer bestimmten Stelle des Programms haben kann, enthalten. Das Belief-Set für unsere Slot-Instanz ist also eine Teilmenge von $\{null, not\_null\}$. Dazu werden Aktionen definiert, die das Belief-Set verändern. Wird z.B. der Pointer mit *NULL* verglichen, ist das Belief-Set vor der Aktion $\{null, not\_null\}$ und nach der Aktion entsprechend $\{null\}$ oder $\{not\_null\}$, je nach dem, wie im Programm verzweigt wird. Wird ein Pointer dereferenziert, ist das Belief-Set nach der Operation $\{not\_null\}$, da Null-Pointer nicht dereferenziert werden können. Zuletzt muss spezifiziert werden, was passiert, wenn zwei Belief-Sets aufeinandertreffen. Für unser Template ist es offensichtlich kein Problem, wenn zwei gleiche Belief-Sets aufeinandertreffen. In diesem Beispiel sieht es allerdings anders aus. In der ersten Zeile wird der Pointer *tty* dereferenziert. Nach dieser Aktion ist das Belief-Set also $\{not\_null\}$. Die Variablendeklaration ändert das Belief-Set nicht. Danach wird abgefragt, ob *tty NULL* ist, oder nicht. Dies führt zu dem Belief-Set $\{null, not\_null\}$ vor dem Vergleich. Nun haben wir zwei sich widersprechende Belief-Sets. Damit wurde ein Fehler gefunden: Entweder ist der Vergleich mit *NULL* überflüssig oder *tty* wird derefenziert, obwohl es ein *NULL*-Pointer sein könnte.

Must-Beliefs können auch über Funktionsgrenzen hinweg überprüft werden. Wenn zwei Funktionen dasselbe Interface implementieren, müssten z.B. ihre Argumente gleich behandelt werden.

Die zweite Gruppe Beliefs sind die „May-Beliefs". Ein Beispiel dafür ist *„Lock* $< l >$ *protects variable* $< v >$". Dass ein bestimmter Lock eine bestimmte Variable schützt, wenn eine entsprechende Stelle im Code gefunden wird, kann der Fall sein. Eventuell steht ein Zugriff auf die Variable aber auch nur zufällig gemischt mit dem Code des eigentlich schützenswerten, kritischen Abschnitts. Wie erkennt man nun, ob wirklich angenommen werden kann, dass ein Lock eine Variable schützt, sodass man bei Verletzungen der Annahme von Fehlern ausgehen kann? Der Trick ist, solche Beliefs auch erst einmal als Must-Beliefs zu behandeln und mitzuzählen, wie oft ein Belief tatsächlich zutrifft und wie oft nicht. Je größer am Ende das Verhältnis zwischen Treffern und Nicht-Treffern ist, desto wahrscheinlicher ist es, dass es sich um eine richtige Annahme handelt [4].

Engler et al. geben an, in verschiedenen Systemen auf diese Art mehrere hundert Fehler gefunden zu haben, räumen aber ein, dass nicht bei allen potentiellen Fehlern genau überprüft wurde, ob die Verletzung des Beliefs tatsächlich ein Fehler ist.

Die größten Pluspunkte für die Meta-Level-Compilation sind, dass man für diese keine Änderungen am Programmcode vornehmen muss und dass auch keine Modelle generiert werden müssen. Der Aufwand ein Programm zu analysieren wächst also nicht mit dessen Größe. Im Gegenteil ist es, wenn man die Analyse über Belief-Templates macht, wahrscheinlich eher vorteilhaft, wenn das Programm viel Code enthält, da sich darin potentiell mehr Beispiele für Beliefs finden lassen.

V. Zusammenfassung

In dieser Arbeit wurden drei unterschiedliche Verfahren zur statischen Fehlererkennung betrachtet, Model-Checking, Fehlererkennung mit Typsystemen und die Meta-Level-Compilation.

Diese Techniken sind zur statischen Fehlererkennung unterschiedlich gut geeignet, abhängig davon, welche Art Fehler man finden möchte. Um logische Fehler im Programm zu finden, hat sich das Model-Checking als geeignet gezeigt. Insbesondere lassen sich damit bestimmte Eigenschaften formal verifizieren, was bei den anderen beiden Techniken nicht der Fall ist. Nachteilig ist allerdings der, auch mit automatischer Modellgenerierung, hohe Aufwand, der zur Analyse betrieben werden muss.

Dagegen ist die Fehlererkennung mit einem statischen Typsystem fast kostenfrei. Ohne Erweiterung lassen sich aber nur sehr einfache Fehler finden. Um Fehler im Ressourcenmanagement zu finden, also Programmierfehler wie „Doppel-*free()*s", kann man auf eine Erweiterung des Typsystems wie das hier betrachtete *Vault* zurückgreifen. Allerdings muss der zu analysierende Code dafür in relativ großem Umfang verändert werden.

Die Meta-Level-Compilation scheint besonders zur Erkennung von klassischen Programmierfehlern wie in den oben

gezeigten Beispielen geeignet zu sein. Außerdem steigt der Aufwand für den Benutzer nicht mit der Größe des Programms, das analysiert werden soll, was sie besonders für große Systeme attraktiv erscheinen lässt.

LITERATUR

[1] G. Behrmann and A. David and K. G. Larsen: *A Tutorial on Uppaal*. 2004, URL: http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf.

[2] J. C. Corbett and M. B. Dwyer and J. Hatcliff and S. Laubach and C. S. Pasareanu and Robby and H. Zheng: *Bandera: Extracting Finite-state Models from Java Source Code*. In: *Proceedings of the 22nd international conference on Software engineering*, 2000.

[3] R. DeLine and M. Fähndrich: *Enforcing High-Level Protocols in Low-Level Software*. In: *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001.

[4] D. Engler and D. Y. Chen and S. Hallem and A. Chou and B. Chelf: *A General Approach to Inferring Errors in Systems Code*. In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[5] D. Engler and B. Chelf and A. Chou and S. Hallem: *Checking system rules using system-specific, programmer-written compiler extensions.* In: *Proceedings of Operating Systems Design and Implementation*, 2000.

[6] P. Kastner and T. Kastner and O. J. L. Riemann and F. H. Wenk: *Betriebsysteme 1: Übungsblatt 1: Lösungsvorschlag*, 2009, unveröffentlicht, auf Anfrage erhältlich.

[7] J. Peleska: *Theory of Reactive Systems*. 2010, URL: http://www.informatik.uni-bremen.de/agbs/lehre/ss10/trs/script/trs_script.pdf.

[8] B. W. Kernighan and D. M. Ritchie: *The C programming language, 2nd ed.*. Prentice Hall, 1988.