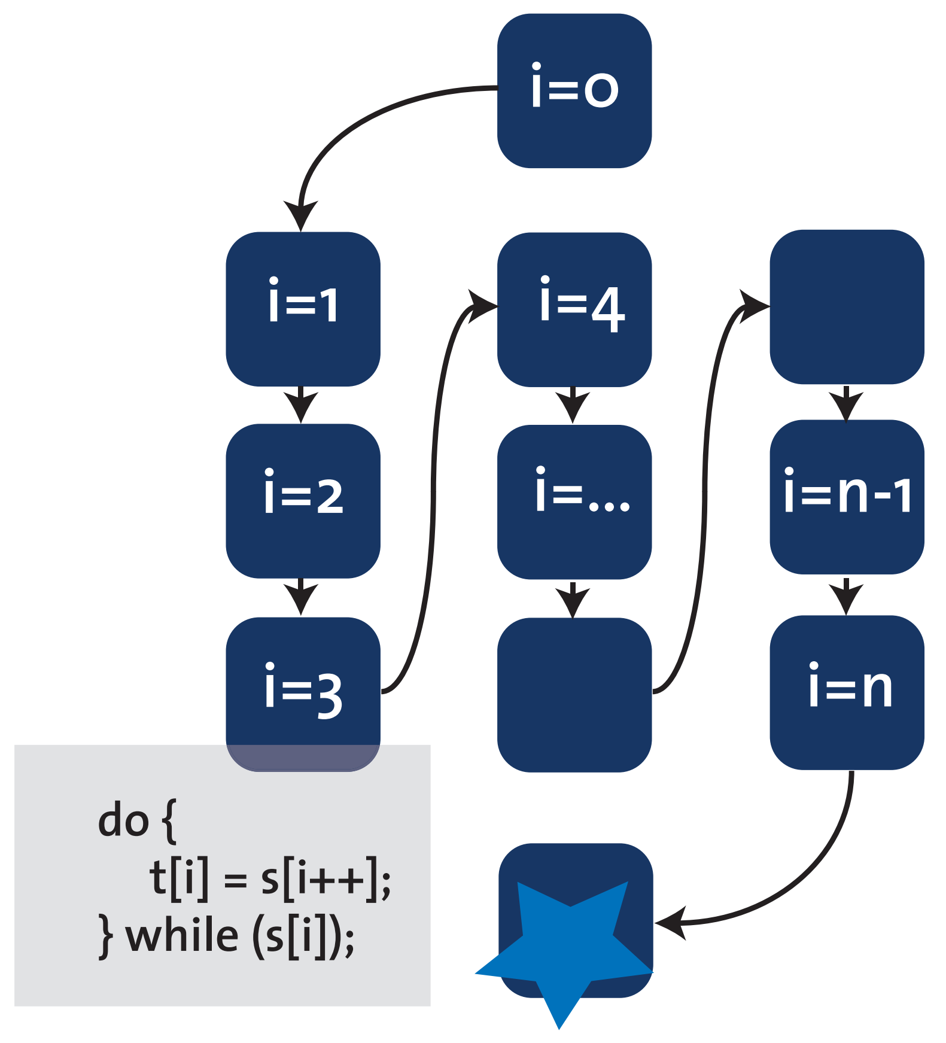
LOOP DETECTION

Advances in Counterexample Guided Abstraction-Refinement

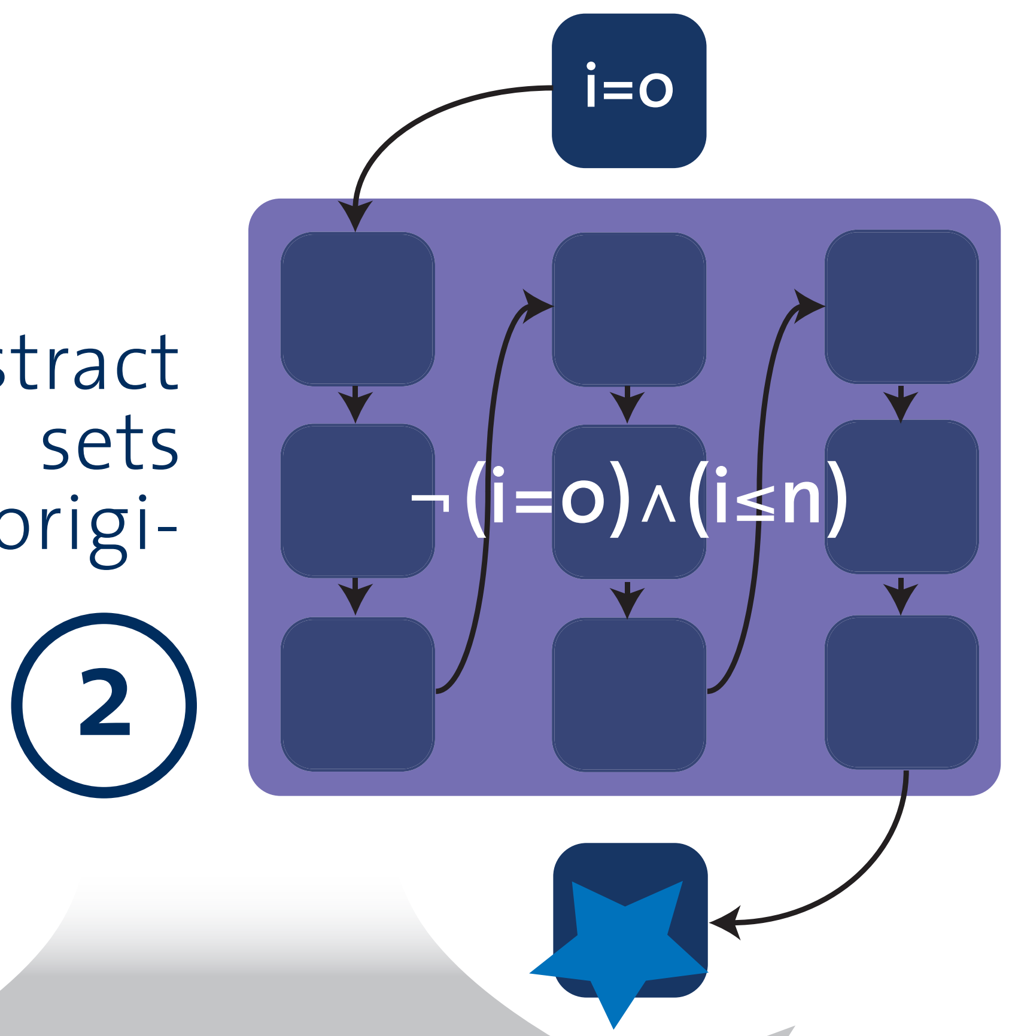
Counterexample-guided abstraction-refinement based on predicate abstraction enables model checking large C programs (such as Windows device drivers). However, the technique is extremely inefficient on programs that contain deep loops. The first intermediate result of our research is a technique that solves this problem.

1 The concrete program contains a buffer overflow that occurs after n iterations of the loop.



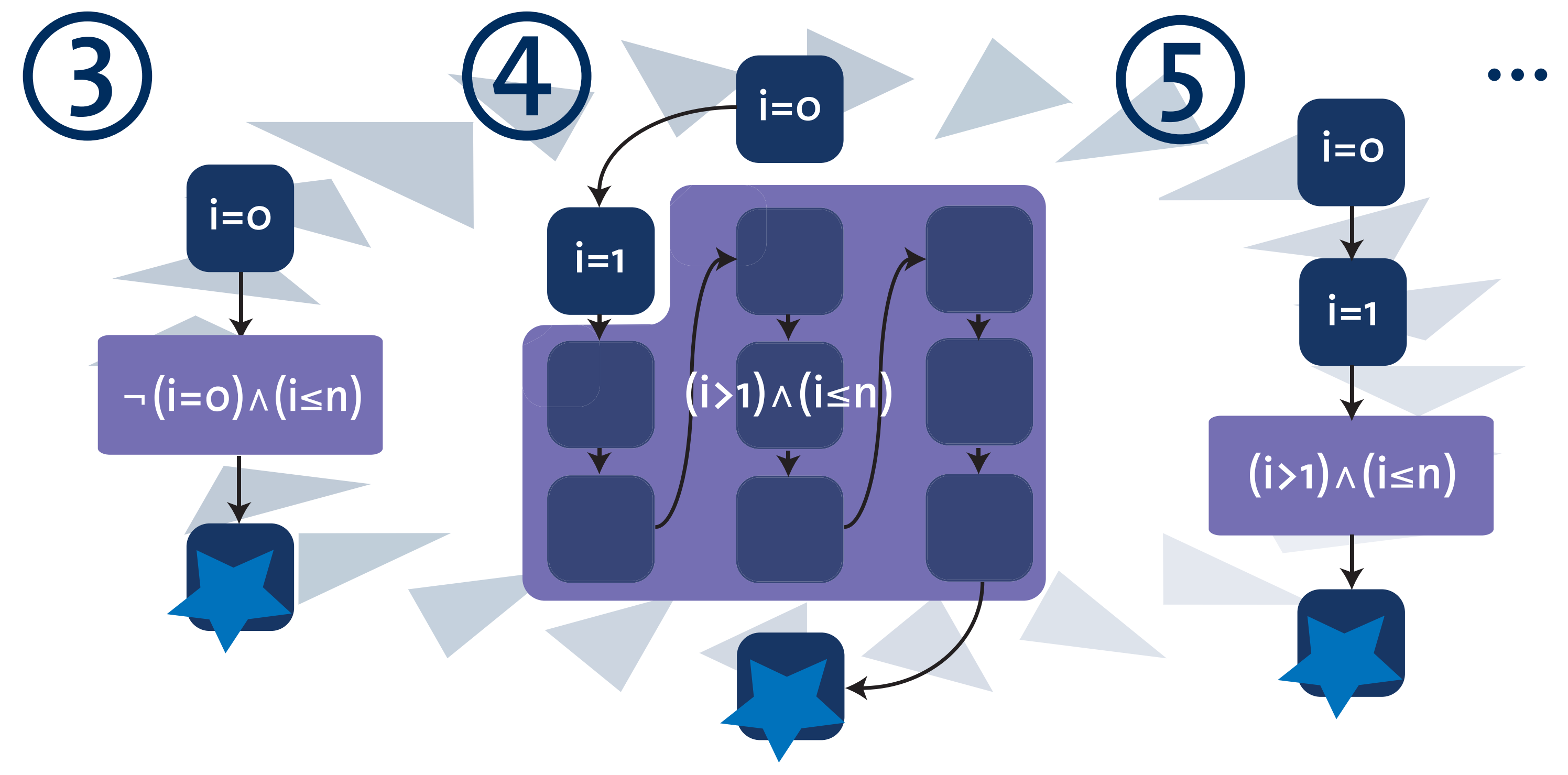
```
do {
    t[i] = s[i++];
} while (s[i]);
```

States of the abstract model represent sets of states of the original program.



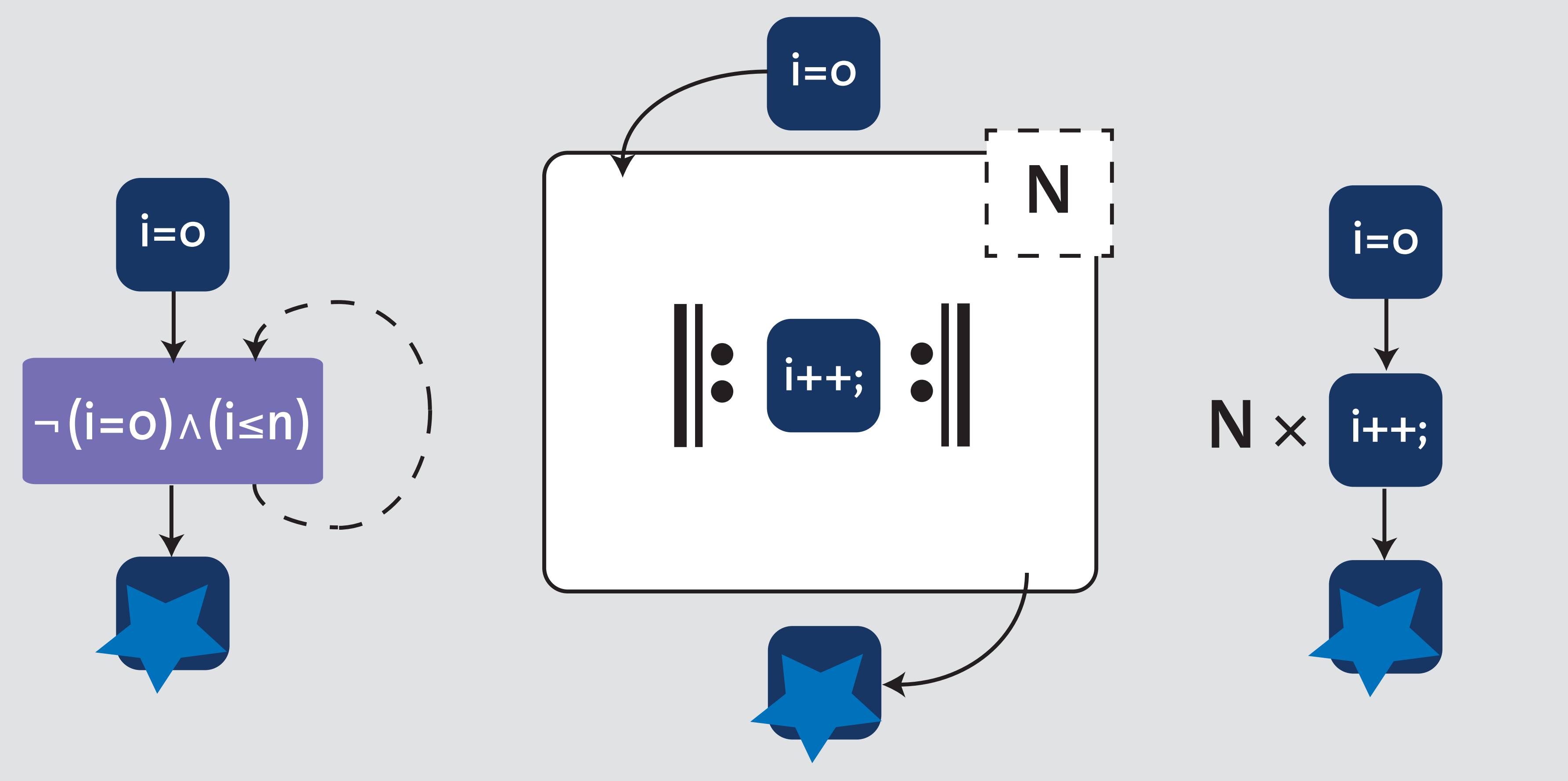
THE TRADITIONAL APPROACH

A model checker exhaustively examines the abstract model and finds an abstract counter-example. The counterexample cannot be replayed on the original program and therefore the abstract model is refined, leading to another spurious counter-example. The abstract model is refined hundreds of times before the buffer overflow is finally detected.



LOOP DETECTION - choosing the short track

Instead of performing the tedious process of iterative unrolling we detect loops in a single step:



Potential loops are detected in the abstract counterexample

The simulation instance is parameterized in the number of iterations

Using a SAT solver, we find the number of iterations that triggers the bug

THE GORY DETAILS

In a post-processing step, we detect potential loops in the abstract counterexample, which consists of a sequence of abstract states (denoted by $\hat{s}_i$):

```
FindLoops($\hat{s}_1, \ldots, \hat{s}_n$)
foreach $i \in \{1, \ldots, n\}$, $j < i$:
    if $\exists \hat{s}'_j, \ldots, \hat{s}'_i . \forall k \in \{j, \ldots, i\} . \ell(\hat{s}'_k) = \ell(\hat{s}_k) \wedge$
        $\forall k \in \{j, \ldots, i-1\} . \hat{s}'_k \xrightarrow{a} \hat{s}'_{k+1} \wedge \hat{s}'_j = \hat{s}_j \wedge \hat{s}'_i \xrightarrow{a} \hat{s}'_j$
    then insert $\|: \hat{s}'_j, \ldots, \hat{s}'_i :\|$
return counterexample $\hat{s}_1, \ldots, \hat{s}_n$ with loops
```

The algorithm searches for transitions that can be taken to jump back to an abstract state that the trace has already traversed. The counterexample is annotated accordingly.
The annotated counterexample is mapped back to the original program. We construct a recurrence equation for the loop induction variable ($i^{(N)}=i^{(N-1)}+1$ in the example from above), put it in its closed form ($i^{(N)}=i^{(0)}+N$) and subsitute the corresponding occurrences in the loop body. Using a SAT-solver, we determine if there is a N that makes this parameterized execution trace feasible. If this is not the case, we proceed with the traditional abstraction-refinement algorithm.
Otherwise, we simulate the unrolled counterexample. Feasible counterexamples constitute bugs and are reported to the user. Spurious counterexamples are used to refine the abstract model the usual way.

WORK IN PROGRESS

With the support of Byron Cook, we have integrated a model checker for asynchronous abstract models into Microsoft's abstraction-refinement toolkit SLAM. Achieving scalability results comparable to sequential analysis is still an on-going effort.

http://www.inf.ethz.ch/personal/daniekro/satabs/

Student: Georg Weissenbacher
Supervisor: Daniel Kroening

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich