Insight: A(nother) Binary Analysis Framework

Emmanuel Fleury
Gérald Point
Aymeric Vincent

<surname.name@labri.fr>

LaBRI, France

Challenges in Analysing Executables:
Scalability, Self-Modifying Code and Synergy
Dagstuhl Seminar 14241, June 10, 2014.

université de BORDEAUX

Overview

1. Motivations and Goals
2. Insight Framework
3. Intermediate Representation
4. CFGRecovery: A Full Disassembler
5. `iii`: Insight Interactive Inspector
6. Perspectives

Overview

1. Motivations and Goals
2. Insight Framework
3. Intermediate Representation
4. CFGRecovery: A Full Disassembler
5. iii: Insight Interactive Inspector
6. Perspectives

Motivations and Goals

Motivations

- Analysis of legacy/off-the-shelf/proprietary software;
- Software reverse-engineering on malware;
- Analysis of software generated by untrusted compilers;
- Analysis of low-level interactions between hardware and OS.

Goals

- Check memory safety of software (reachability, liveness, fairness);
- Detect potential security flaws (mediation, non-interference, ...);
- Behavioral analysis of unknown software (malware detection);
- Full verification of small binaries with strong hypotheses (nuclear power-plant, aircraft, spaceships, transportation systems).

Overview

1. Motivations and Goals
2. Insight Framework
   - Insight Features
   - Insight Framework
   - Insight Library Modules
   - Potential Tools
3. Intermediate Representation
4. CFGRecovery: A Full Disassembler
5. iii: Insight Interactive Inspector
6. Perspectives

Features

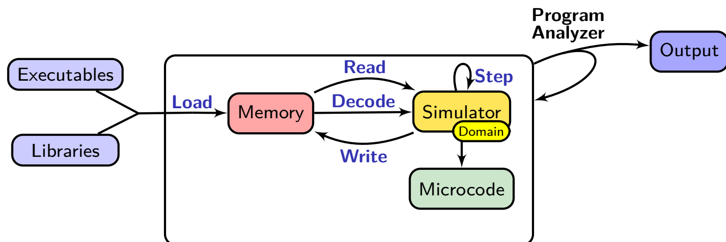
- Running on **UNIXish** platforms.
- Programmed in **C++ language**.
- Available under a **BSD 2-clause** license.

- **Library of modules** to build **prototypes** and **real tools**...

- Executable file format (`GNU binutils`): **ELF**, *PE*, *Mach-O*.
- CPU Architectures: **i386**, **amd64**, **msp430**, *arm*, *sparc*.
- SMT-solvers: **MathSAT5** and *Z3*.

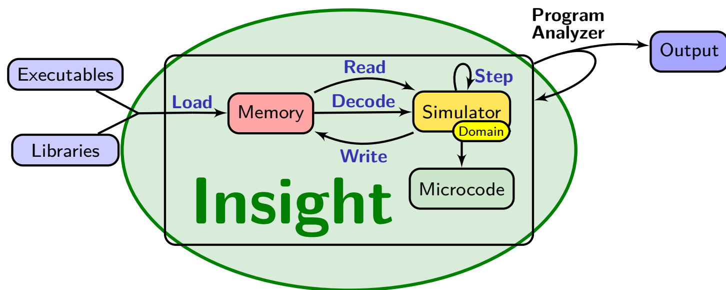
- Intermediate Representation: **Microcode**
  (Model in which any architecture could be translated to);
- Domains for Variables: **Concrete**, **Formula**, *Intervals*, *Sets*;
- Analysis: **Slicing**, **Disassembly methods**, ...

Insight Framework



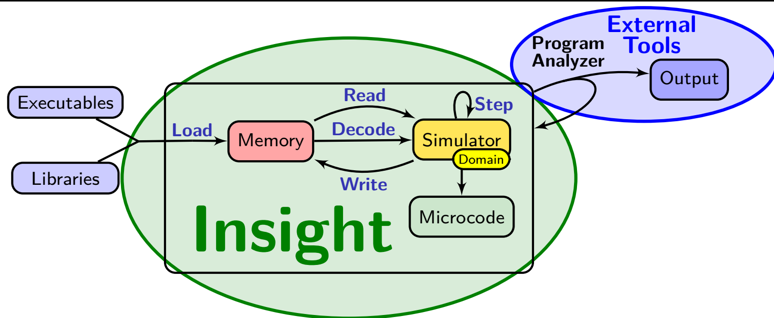
- **Loader**: Open the input file, parse the **meta-data** enclosed in the binary file and extract the **code** to be mapped in memory.
- **Decoder**: Given a **sequence of bytes**, translate it into a **semantics aware instruction set** (microcode).
- **Simulator**: Execute **one step** of the program **according to the semantics**.
- **Microcode**: A **model** representing the **semantics of the binary program**.
- **Domain**: The **concrete/abstract domain** used by the simulation.

Insight Framework



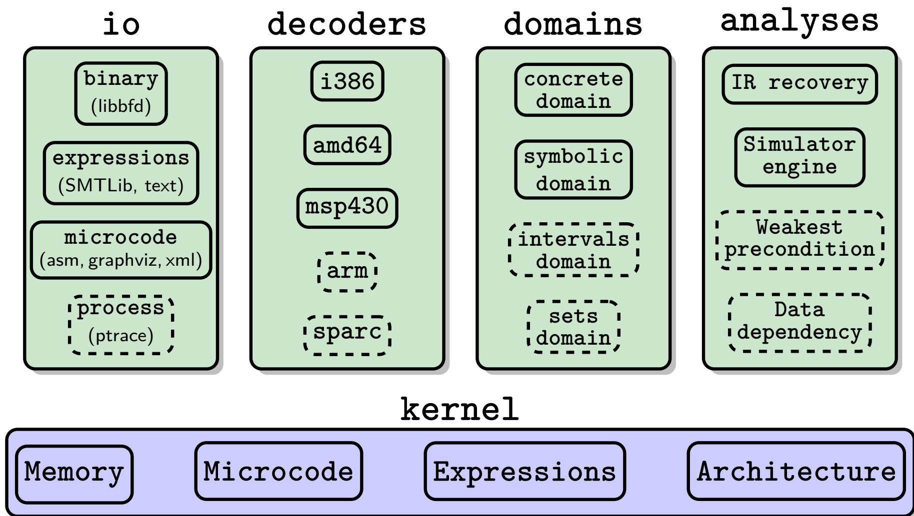
- **Loader**: Open the input file, parse the **meta-data** enclosed in the binary file and extract the **code** to be mapped in memory.
- **Decoder**: Given a **sequence of bytes**, translate it into a **semantics aware instruction set** (microcode).
- **Simulator**: Execute **one step** of the program **according to the semantics**.
- **Microcode**: A **model** representing the **semantics of the binary program**.
- **Domain**: The **concrete/abstract domain** used by the simulation.

Insight Framework



- **Loader**: Open the input file, parse the **meta-data** enclosed in the binary file and extract the **code** to be mapped in memory.
- **Decoder**: Given a **sequence of bytes**, translate it into a **semantics aware instruction set** (microcode).
- **Simulator**: Execute **one step** of the program **according to the semantics**.
- **Microcode**: A **model** representing the **semantics of the binary program**.
- **Domain**: The **concrete/abstract domain** used by the simulation.

Insight Library Modules



Potential Tools

- **Disassembler**: Combination of a **decoder** and a **strategy** to gather microcode as close as possible to the whole program.
- **Symbolic Debugger**: In a binary analysis context, a **symbolic debugger** is a debugger that follows the execution of the program along the **microcode**. Note that the user can choose among several domains (concrete, bit-vector logic formula, . . .).
- **Decompiler**: Translate the **assembly code** into a **high-level language code** with variables, functions and more (modules, objects, classes, . . .).
- **Verifier**: Check if the binary program verifies some properties such as accessibility, safety or fairness.
- **Test Synthesizer**: Synthesize test cases with a good coverage of the program only based on the exploration of the binary.
- **Others. . .**

Potential Tools

- **Disassembler**: Combinatio **CFGRecovery** **strategy** to gather microcode as close as possi am.
- **Symbolic Debugger**: In a binary analysis context, a **symbolic debugger** is a debugger that follows the execution of the program along the **microcode**. Note that the user can choose among several domains (concrete, bit-vector logic formula, . . .).
- **Decompiler**: Translate the **assembly code** into a **high-level language code** with variables, functions and more (modules, objects, classes, . . .).
- **Verifier**: Check if the binary program verifies some properties such as accessibility, safety or fairness.
- **Test Synthesizer**: Synthesize test cases with a good coverage of the program only based on the exploration of the binary.
- **Others. . .**

Potential Tools

- **Disassembler**: Combinatio **CFGRecovery** **strategy** to gather microcode as close as possi am.
- **Symbolic Debugger**: In a binary analysis context, a **symbolic debugger** is a debugger that follows the execut **iii** e program along the **microcode**. Note that the user can choose am al domains (concrete, bit-vector logic formula, . . .).
- **Decompiler**: Translate the **assembly code** into a **high-level language code** with variables, functions and more (modules, objects, classes, . . .).
- **Verifier**: Check if the binary program verifies some properties such as accessibility, safety or fairness.
- **Test Synthesizer**: Synthesize test cases with a good coverage of the program only based on the exploration of the binary.
- **Others. . .**

Overview

1 Motivations and Goals

2 Insight Framework

3 Intermediate Representation
- Insight Microcode
- Expressions
- Basic Syntax
- Transitions

4 CFGRecovery: A Full Disassembler

5 iii: Insight Interactive Inspector

6 Perspectives

Insight Microcode

Microcode is our **intermediate representation** used to record all the information collected about the **semantics** of the analyzed program.

Microcode representation is an automaton where

- **Nodes** are labelled by **memory locations**;
- **Edges** contain **guards** and **instructions**;
- **Guards** and **instructions** contain **expressions**.

Nodes and edges can be annotated by arbitrary objects

- Assembly instructions which originated this microcode;
- Procedure calls/returns known or found;
- Procedure start/end;
- Higher-level constructs discovered;
- ...

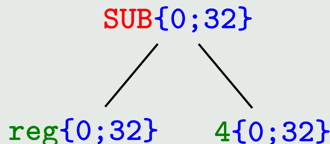
Expressions

“register minus four” (`reg - 4`)

Text-form

`(SUB reg{0;32} 4{0;32}){0;32}`

Tree-form



- Operate on bit-vector arithmetics (`QF_AUFBV`);
- Are used in instructions and guards;
- Are very expressive:
  - Arithmetic operators (`ADD`, `SUB`, `MUL`, `AND`, ...);
  - Operate on registers and immediate values;
  - Concatenation, sign extensions, bit reversal, ...
  - Every expression node can extract a sub-bitvector.

Expressions

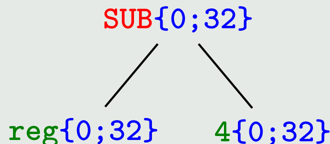
"register minus four" (`reg - 4`)

Text-form

`(SUB reg{0;32} 4{0;32}){0;32}`

Operator

Tree-form



- Operate on bit-vector arithmetics (`QF_AUFBV`);
- Are used in instructions and guards;
- Are very expressive:
  - Arithmetic operators (`ADD`, `SUB`, `MUL`, `AND`, ...);
  - Operate on registers and immediate values;
  - Concatenation, sign extensions, bit reversal, ...
  - Every expression node can extract a sub-bitvector.

Expressions

"register minus four" (`reg - 4`)

Text-form

`(SUB reg{0;32} 4{0;32}){0;32}`

Operator — Arguments

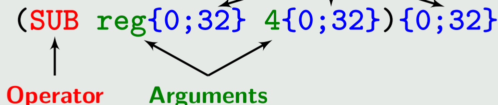
Tree-form

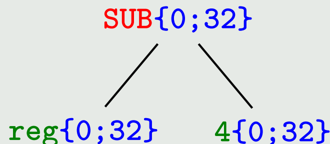
```
        SUB{0;32}
        /       \
reg{0;32}     4{0;32}
```

- Operate on bit-vector arithmetics (`QF_AUFBV`);
- Are used in instructions and guards;
- Are very expressive:
  - Arithmetic operators (`ADD`, `SUB`, `MUL`, `AND`, . . .);
  - Operate on registers and immediate values;
  - Concatenation, sign extensions, bit reversal, . . .
  - Every expression node can extract a sub-bitvector.

Expressions

"register minus four" (reg - 4)





- Operate on bit-vector arithmetics (`QF_AUFBV`);
- Are used in instructions and guards;
- Are very expressive:
  - Arithmetic operators (`ADD`, `SUB`, `MUL`, `AND`, ...);
  - Operate on registers and immediate values;
  - Concatenation, sign extensions, bit reversal, ...
  - Every expression node can extract a sub-bitvector.

Basic Syntax (1/2)

Addresses: One kind of address for data and one for microcode instructions. These latter addresses are composed of a memory location and a micro-location.

```
       data:   0xdeadbeef
instruction:   (0xdeadbeef, 2)
```

Dereferencement: Accessing memory address 'addr' is denoted '[addr]'.

```
[0xdeadbeef]{0;32}
```

Random: Represents a random value.

```
%reg := RND
```

Annotation: Annotations are key-value pairs displayed like this: '@{lbl:=annotation}@'.

```
@{asm := call 0x1235}@
```

Basic Syntax (2/2)

Skip: A "*do nothing*" statement.

```
Skip
```

Assignment: Simply denoted by '`l-value := expr`'.

```
[%esp{0;32}]{0;32} := 0x5{0;32}
```

Jump: Simply denoted by '`Jmp expr`'.

```
Jmp [%eax{0;32}]{0;32}
```

Transitions

Static Jump

```
[addr,uaddr] <<guard>> stmt --> (addr,uaddr);
```

```
[0x2,5] %reg{0;8} := (SUB %reg{0;8} 0x1{0;8}){0;8} --> (0x2,2);
```

Dynamic Jump

```
[addr,uaddr] <<guard>> Jmp --> expr;
```

```
[0x5,2] << (EQ %reg{0;32} 0x0{0;32}) >> Jmp --> [0x12{0;32}]{0;32};
```

Example (Text-form)

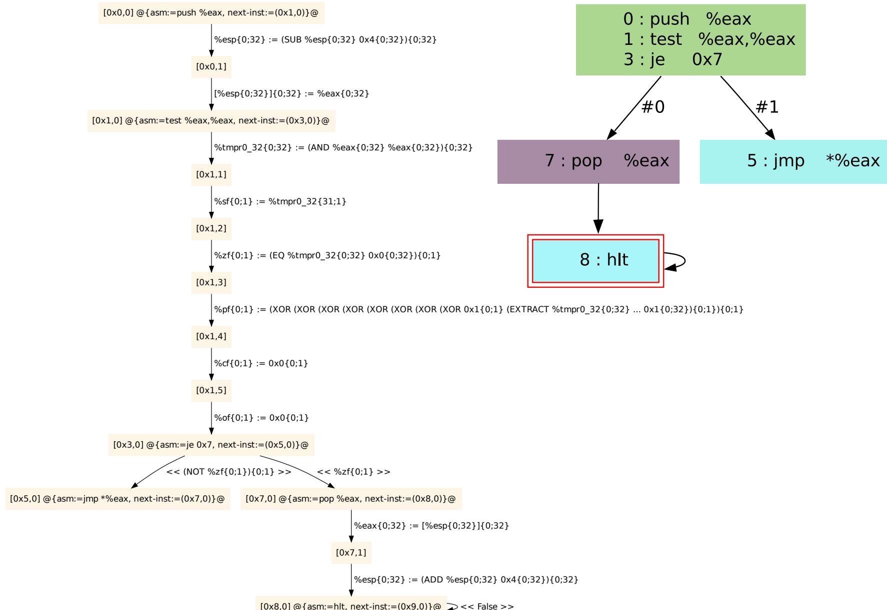
Assembly

```
00000000 <start>:
   0: 50                    push   %eax
   1: 85 c0                 test   %eax,%eax
   3: 74 02                 je     0x7
   5: ff e0                 jmp    *%eax
   7: 58                    pop    %eax
   8: f4                    hlt
```

Microcode

```
[0x0,0] %esp{0;32} := (SUB %esp{0;32} 4{0;32}){0;32};    --> (0x0,1)
[0x0,1] [%esp{0;32}]{0;32} := %eax{0;32} ;               --> (0x1,0)
[0x1,0] %tmp{0;32} := (AND %eax{0;32} %eax{0;32}){0;32}; --> (0x1,1)
[0x1,1] %nf{0;1} := (LT %tmp{0;32} 0{0;32}){0;32};       --> (0x1,2)
[0x1,2] %zf{0;1} := (EQ %tmp{0;32} 0{0;32}){0;32};       --> (0x1,3)
[0x1,3] %pf{0;1} := (XOR (XOR %tmp{0;1} %tmp{1:1}...     --> (0x1,4)
[0x1,4] %cf{0;1} := 0{0;1};                              --> (0x1,5)
[0x1,5] %of{0;1} := 0{0;1};                              --> (0x3,0)
[0x3,0] ...
```

Example (Graph-form)



Overview

1. Motivations and Goals
2. Insight Framework
3. Intermediate Representation
4. **CFGRecovery: A Full Disassembler**
5. iii: Insight Interactive Inspector
6. Perspectives

Goals and Features

Try to recover as much as possible of the IR in a fully automated mode.

Disassembly Strategies

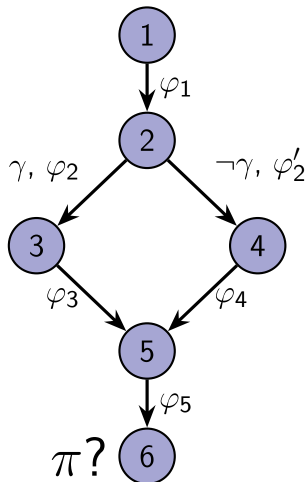
- Linear Sweep;
- Recursive Traversal;
- Flood Traversal;
- **Symbolic Execution**.

Execution Domains

- Concrete
- **Symbolic** (Formula)
- Sets (Untested)

Path Formula

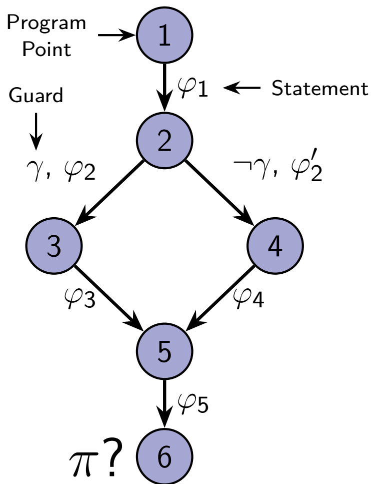
How to compute all the possible valuations that can reach program point 6 ?



$$\pi = \varphi_1 \wedge ((\gamma \wedge \varphi_2 \wedge \varphi_3) \vee (\neg\gamma \wedge \varphi_2' \wedge \varphi_4)) \wedge \varphi_5$$

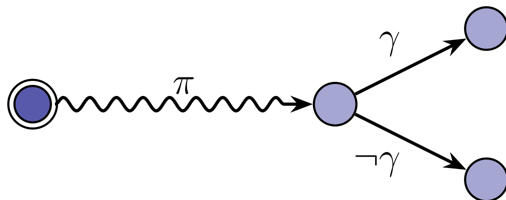
Path Formula

How to compute all the possible valuations that can reach program point 6 ?



$$\pi = \varphi_1 \wedge ((\gamma \wedge \varphi_2 \wedge \varphi_3) \vee (\neg\gamma \wedge \varphi'_2 \wedge \varphi_4)) \wedge \varphi_5$$

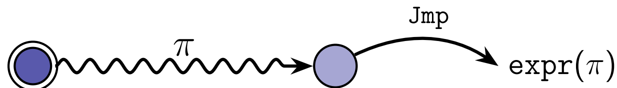
Conditional Static Jumps



$$SAT(\pi \wedge \gamma) = \begin{cases} \textbf{Satisfiable} \text{ or } \textbf{Unconclusive}\text{, add target to worklist;} \\ \textbf{Unsatisfiable}\text{, ignore.} \end{cases}$$

$$SAT(\pi \wedge \neg\gamma) = \begin{cases} \textbf{Satisfiable} \text{ or } \textbf{Unconclusive}\text{, add target to worklist;} \\ \textbf{Unsatisfiable}\text{, ignore.} \end{cases}$$

Dynamic Jumps



$$\mathit{SOLVE}(\pi \wedge \mathtt{expr}) = \begin{cases} \mathtt{\{0xabcdef, \ ...\}}, \text{ add target to worklist;} \\ \textbf{Empty Set}, \text{ should be impossible.} \end{cases}$$

Symbolic Execution Disassembler

Algorithm 1 Symbolic Execution Disassembler

Require: W (worklist of program points), MC (microcode) and Π (path formula associated to each encountered program point).

function SYMBOLICEXECUTION($Prog = (M_0, a_0)$)
- $W \leftarrow a_0$, and $MC, \Pi \leftarrow \emptyset$
- **while** ($W \neq \emptyset$) **do**
  - $a \leftarrow getItem(W)$
  - $Succs \leftarrow getSuccessors(a)$
  - **for** $succ \in Succs$ **do**
    - **if** ($succ.isStatic() \wedge SAT(\Pi(a, succ))$) **then**
      - $W, MC \leftarrow succ$, $\Pi \leftarrow \Pi(a, succ)$
    - **else if** ($succ.isDynamic()$) **then**
      - $W, MC \leftarrow SOLVE(\Pi(a, succ))$, $\Pi \leftarrow \Pi(SOLVE(a, succ))$
    - **end if**
  - **end for**
- **end while**

end function

SMT-Solvers

SMT-solvers combine the efficiency of a SAT-solver and theory-specific decision procedure. For example, the following formula combines propositional logic with quantifier free bit-vectors with arrays and uninterpreted functions (`QF_AUFBV`):

$$a \wedge ((x + y < 0) \vee \neg a \wedge ((x = 1) \vee b), \; a, b \in \mathbb{B}, \; x, y \in \mathbf{BV}$$

Solving procedure

SMT-Solvers

SMT-solvers combine the efficiency of a SAT-solver and theory-specific decision procedure. For example, the following formula combines propositional logic with quantifier free bit-vectors with arrays and uninterpreted functions (`QF_AUFBV`):

$$a \wedge ((x + y < 0) \vee \neg a \wedge ((x = 1) \vee b), \; a, b \in \mathbb{B}, \; x, y \in \mathbf{BV}$$

Solving procedure

1. Identify the boolean parts of the formula;

SMT-Solvers

SMT-solvers combine the efficiency of a SAT-solver and theory-specific decision procedure. For example, the following formula combines propositional logic with quantifier free bit-vectors with arrays and uninterpreted functions (`QF_AUFBV`):

$$a \wedge ((x + y < 0) \vee \neg a \wedge ((x = 1) \vee b), \; a, b \in \mathbb{B}, \; x, y \in \mathbf{BV}$$

Solving procedure

1. Identify the boolean parts of the formula;
2. Identify the `QF_AUFBV` parts of the formula;

SMT-Solvers

SMT-solvers combine the efficiency of a SAT-solver and theory-specific decision procedure. For example, the following formula combines propositional logic with quantifier free bit-vectors with arrays and uninterpreted functions (`QF_AUFBV`):

$$a \wedge (\underbrace{(x + y < 0)}_{c} \vee \neg a \wedge (\underbrace{(x = 1)}_{d} \vee b), \quad a, b \in \mathbb{B}, \quad \underset{c, d \in \mathbb{B}}{x, y \in \mathbf{BV}}$$

Solving procedure

1. Identify the boolean parts of the formula;
2. Identify the `QF_AUFBV` parts of the formula;
3. Substitute the `QF_AUFBV` terms by boolean clauses;

SMT-Solvers

SMT-solvers combine the efficiency of a SAT-solver and theory-specific decision procedure. For example, the following formula combines propositional logic with quantifier free bit-vectors with arrays and uninterpreted functions (`QF_AUFBV`):

$$a \wedge (\underbrace{(x + y < 0)}_{c} \vee \neg a \wedge (\underbrace{(x = 1)}_{d} \vee b), \quad a, b \in \mathbb{B}, \quad x, y \in \mathbf{BV}$$

$$c, d \in \mathbb{B}$$

Solving procedure

1. Identify the boolean parts of the formula;
2. Identify the `QF_AUFBV` parts of the formula;
3. Substitute the `QF_AUFBV` terms by boolean clauses;
4. Check satisfiability of the formula with a SAT-solver (enumerate all possible boolean models);

SMT-Solvers

SMT-solvers combine the efficiency of a SAT-solver and theory-specific decision procedure. For example, the following formula combines propositional logic with quantifier free bit-vectors with arrays and uninterpreted functions (`QF_AUFBV`):

$$a \wedge (\underbrace{(x + y < 0)}_{c} \vee \neg a \wedge (\underbrace{(x = 1)}_{d} \vee b), \quad a, b \in \mathbb{B}, \quad \underset{c, d \in \mathbb{B}}{x, y \in \mathbf{BV}}$$

Solving procedure

1. Identify the boolean parts of the formula;
2. Identify the `QF_AUFBV` parts of the formula;
3. Substitute the `QF_AUFBV` terms by boolean clauses;
4. Check satisfiability of the formula with a SAT-solver (enumerate all possible boolean models);
5. If some boolean models are found satisfiable, check if they can also be instanciated in the `QF_AUFBV` logic;

SMT-Solvers

SMT-solvers combine the efficiency of a SAT-solver and theory-specific decision procedure. For example, the following formula combines propositional logic with quantifier free bit-vectors with arrays and uninterpreted functions (`QF_AUFBV`):

$$a \wedge (\underbrace{(x + y < 0)}_{c} \vee \neg a \wedge (\underbrace{(x = 1)}_{d} \vee b), \quad a, b \in \mathbb{B}, \quad \underset{c, d \in \mathbb{B}}{x, y \in \mathbf{BV}}$$

Solving procedure

1. Identify the boolean parts of the formula;
2. Identify the `QF_AUFBV` parts of the formula;
3. Substitute the `QF_AUFBV` terms by boolean clauses;
4. Check satisfiability of the formula with a SAT-solver (enumerate all possible boolean models);
5. If some boolean models are found satisfiable, check if they can also be instanciated in the `QF_AUFBV` logic;
6. Return the result.

Current Issues & Future Work

Current Issues

- Formula Simplification;
- Handling self-modifying code;
- Improve speed efficiency.

Future Work ?

- Loops summarization;
- Variables and types recovery;
- Interprocedural analysis;
- Multiple memory arrays;
- Scaling up to big programs;
- ...

Overview

1 Motivations and Goals

2 Insight Framework

3 Intermediate Representation

4 CFGRecovery: A Full Disassembler

5 `iii`: Insight Interactive Inspector

6 Perspectives

Pynsight (Python Layer)

Gives access to libinsight

Coarse grained

- "only" libinsight type visible: program
- rest of the API communicates through strings/ints
- we want to keep (most of) the code written in C++
- still leverages the power of Python (easy loops, ...)

iii: Insight Interactive Inspector

- Written in python using pynsight
- Symbolic simulator presented as a debugger
- Follows a trace, and accumulates the microcode
- Can start other traces enriching the same microcode
- SMT solver guarantees trace is executable

Features of iii

Symbolic debugger for assembly and microcode levels.

- Interactive step by step symbolic execution for assembly or microcode;
- Display the current state of the intermediate representation in assembly or microcode;
- Give access to symbolic memory (read/write);
- Allow to create and manage breakpoints/watchpoints on the microcode;
- Fully scriptable debugger (Python);
- Load/save annotated microcode

Demo (iii)

Overview

1. Motivations and Goals
2. Insight Framework
3. Intermediate Representation
4. CFGRecovery: A Full Disassembler
5. iii: Insight Interactive Inspector
6. Perspectives

Perspectives

Features

- Architectures: i386, amd64, msp430;
- Linear sweep, Recursive traversal;
- Symbolic execution with SMT-solver;
- Procedure stubbing;
- Interactive symbolic debugger.

Future Work

- Architectures: ARM, Sparc, . . . ;
- Formula simplification;
- Loop summarization;
- Dynamic library loader.

Also currently looking for a **PostDoc/Research Engineer** !

10-16 months position in Bordeaux, France
Requires skills in C++ and assembly, knowledge in program verification.

Questions ?

Website	`https://insight.labri.fr`
SVN	`https://insight.labri.fr/svn/trunk`
GitHub	`https://github.com/perror/insight`