BINSEC:
Binary-level Semantic Analysis to the Rescue

Sébastien Bardin
joint work with
Richard Bonichon, Robin David, Adel Djoudi, Benjamin Farinier, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dihn Ta, Franck Védrine

CEA LIST (Paris-Saclay, France)
About the BINSEC project

A research project:

- funded by ANR (2013-2017)
- axis 1 (security) and 2 (software engineering)
- formal techniques for binary-level security analysis

Partners: CEA (coordinator), Airbus Group, INRIA Bretagne Atlantique, Université Grenoble Alpes, Université de Lorraine

People: Sébastien Bardin, Frédéric Besson, Sandrine Blazy, Guillaume Bonfante, Richard Bonichon, Robin David, Adel Djoudi, Benjamin Farinier, Josselin Feist, Colas Le Guernic, Jean-Yves Marion, Laurent Mounier, Marie-Laure Potet, Than Dihnh Ta, Franck Védrine, Pierre Wilke, Sara Zennou

Platform: CEA, Université Grenoble Alpes
Binary-level security analysis

- many applications, many challenges
- syntactic and dynamic methods are not sufficient

Semantic approaches can help!

- semantic exploration, semantic disassembly
- yet, still hard to design

The BINSEC Platform [CEA & Uni. Grenoble Alpes]

- open source, dual goal:
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers

- current: multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerability analysis and deobfuscation

- still young: beta-version just released [http://binsec.gforge.inria.fr/]

BINSEC team
CEA LIST, Software Safety & Security Lab

- rigorous tools for building high-level quality software
- 2nd part of V-cycle
- automatic software analysis
- mostly source code
About formal verification

- Between Software Engineering and Theoretical Computer Science
- Goal = proves correctness in a mathematical way

Key concepts: $M \models \varphi$
- $M$: semantic of the program
- $\varphi$: property to be checked
- $\models$: algorithmic check

Kind of properties
- absence of runtime error
- pre/post-conditions
- temporal properties
Industrial reality in some key areas, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

Ex : Airbus

Verification of

- runtime errors [Astrée]
- functional correctness [Frama-C *]
- numerical precision [Fluctuat *]
- source-binary conformance [CompCert]
- ressource usage [Absint]

* : by CEA DILS/LSL
From (a logician’s) dream to reality

**Industrial reality** in some **key areas**, especially safety-critical domains
- hardware, aeronautics [*airbus*], railroad [*metro 14*], smartcards, drivers [*Windows*], certified compilers [*CompCert*] and OS [*Sel4*], etc.

---

**Ex: Microsoft**

Verification of drivers [*SDV*]
- conformance to MS driver policy
- home developers
- and third-party developers

*Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we’re building tools that can do actual proof about the software and how it works in order to guarantee the reliability.*

- Bill Gates (2002)
Outline

- Preamble
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- Conclusion
Benefits of binary-level analysis

Binary-level software analysis

Model

Source code

int foo(int x, int y) {
    int k = x;
    int c = y;
    while (c > 0) do {
        k++;
        c--;
    }
    return k;
}

Assembly

Executable

/start:
    load A 100
    add B A
    cmp B 0
    jle label

label:
    move @100 B

ABFFF780BD70696CA101001BDE45
145634789234ABFFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
Benefits of binary-level analysis

What for? (1)

COTS

How much do you trust your external components?
Benefits of binary-level analysis

What for? (2)

How much do you trust your compiler?
Security bug introduced by a non-buggy compiler

```c
void getPassword(void) {
    char pwd [64];
    if (GetPassword(pwd,sizeof(pwd))) {
        /* checkpassword */
    }
    memset(pwd,0,sizeof(pwd));
}
```

- Optimizing compilers may remove dead code
- `pwd` never accessed after `memset`
- Thus can be safely removed
- And allows the password to stay longer in memory

Mentioned in OpenSSH CVE-2016-0777
Benefits of binary-level analysis

What for? (3)

Is it Stuxnet?
Outline

- Preambule
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- Conclusion
Several major security analyses are performed at byte-level

- vulnerability analysis [exploit finding]
- malware dissection and detection [deobfuscation]

State-of-the-technique

- very skilled experts, many efforts and basic tools
- dynamic analysis: gdb, fuzzing [easy to miss behaviours]
- syntactic analysis: objdump, IDA Pro [easy to get fooled]
Several major security analyses are performed at byte-level

- vulnerability analysis [exploit finding]
- malware dissection and detection [deobfuscation]

State-of-the-technique

- very skilled experts, many efforts and basic tools
- dynamic analysis: gdb, fuzzing [easy to miss behaviours]
- syntactic analysis: objdump, IDA Pro [easy to get fooled]

state-of-the-art tools are not enough!
Challenges of binary-level analysis

Challenge: correct disassembly

Input
- an executable code (array of bytes)
- an initial address
- a basic decoder: file $\times$ address $\mapsto$ instruction $\times$ size

Output: (surapproximation of) the program Control-Flow Graph
- problem: successors of jmp eax?
Ex: IDA is fooled by simple syntactic tricks

With IDA
Challenges of binary-level analysis

Even worse: obfuscated code

Understand or recognize malware despite obfuscation

- self-modifying code, virtual machines
- opaque predicates, stack tampering, etc.

Context: x86-malware

A common protection scheme for malware
a SillyFDC run

Self-modifying program schema
Challenges of binary-level analysis

Challenges: vulnerabilities

**Use-after-free (UaF) – CWE-416**

- *dangling pointer* on deallocated-then-reallocated memory
- may lead to arbitrary data/code read, write or execution
- standard vulnerability in C/C++ applications (e.g. web browsers)
  - firefox (CVE-2014-1512), chrome (CVE-2014-1713)

```c
char *login, *passwords;
login=(char *) malloc (...);
[...]
free(login); // login is now a dangling pointer
[...]
passwords=(char *) malloc (...); // may re-allocate memory of *login
[...]
printf("%s\n", login); // security threat: may print the passwords!
```
Challenges of binary-level analysis

Limits of dynamic analysis

Find a needle in the heap!

- sequence of events, importance of aliasing
- strongly depend on implem of malloc and free
Outline

- Preambule
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- Conclusion
Semantic tools help make sense of binary

- Develop the next generation of binary-level tools!
- motto: leverage formal methods from safety critical systems

Challenges

- source-level $\mapsto$ binary-level
- safety $\mapsto$ security
- many (complex) architectures
Binary-level semantic approaches

BINSEC approach

leverage powerful methods from formal software analysis

pragmatic formal methods (combination, tradeoffs, etc.)

common basic analysis + dedicated analysis (vuln., malware)
Binary-level semantic approaches

**Focus**: modelling

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mod R/M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scale Index Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Up to four prefixes of 1 byte each (optional)
- 1-, 2-, or 3-byte opcode
- 1 byte (if required)
- 1 byte (if required)
- Address displacement of 1, 2, or 4 bytes or none
- Immediate data of 1, 2, or 4 bytes or none

Example of x86

- more than 1,000 instructions
  - ≈ 400 basic
  - + float, interrupts, mmx
- many side-effects
- error-prone decoding
  - addressing modes, prefixes, ...
Binary-level semantic approaches

Focus: modelling

Intermediate Representation [cav11]
- architecture independent
- (really) reduced set of instructions
  - 9 instructions, less than 30 operators
- simple, clear semantic, no side-effect

- \( \text{lhs} := \text{rhs} \)
- \( \text{goto addr, goto expr} \)
- \( \text{ite(\text{cond})? goto addr : goto addr'} \)
- \( \text{assume, assert, nondet, malloc, free} \)
Binary-level semantic approaches

**x86 front-end**

---

**Instruction Prefixes**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to four prefixes of 1 byte each (optional)</td>
<td>1, 2, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

---

**Example:**

```
81 c3 57 1d 00 00 \Rightarrow ADD EBX 1d57
```

---

```
(0x29e,0) tmp := EBX + 7511;
(0x29e,1) OF := (EBX{31,31} = 7511{31,31}) && (EBX{31,31} <> tmp{31,31});
(0x29e,2) SF := tmp{31,31};
(0x29e,3) ZF := (tmp = 0);
(0x28e,4) AF := ((extu (EBX{0,7}) 9) + (extu 7511{0,7} 9)){8,8};
(0x29e,6) CF := ((extu EBX 33) + (extu 7511 33)){32,32};
(0x29e,7) EBX := tmp; goto (0x2a4,0)
```
Binary-level semantic approaches

Semantic disassembly

- simple obfuscation confuses SOA disassemblers such as IDA
- ... because they rely on syntax
- semantic techniques complement and strengthen these approaches
Binary-level semantic approaches

Semantic disassembly (2)

With IDA
Binary-level semantic approaches

Semantic disassembly (2)

With IDA + BINSEC

0x4013e0 push %ebp
0x4014e1 mov %esp,%ebp
...
0x401430 mov 0x4(%esp),%eax
0x401434 shl $0x2,%eax
0x40143e add $0x40a064,%eax
0x40143c mov (%eax),%eax
0x401441 mov %eax,%ecx
0x401446 mov %ecx,%eax
0x40144b jmp *%eax

0x4013e0 push %ebp
0x4014e1 mov %esp,%ebp
...
0x401430 mov 0x4(%esp),%eax
0x401434 shl $0x2,%eax
0x40143e add $0x40a064,%eax
0x40143c mov (%eax),%eax
0x401441 mov %eax,%ecx
0x401446 mov %ecx,%eax
0x40144b jmp *%eax
Binary-level semantic approaches

Semantic disassembly: keys

Generalize constant propag

Framework: abstract interpretation

- notion of abstract domain $\bot, \top, \sqcup, \sqcap, \subseteq, \text{eval}#$
- more or less precise domains
  - intervals, polyhedra, etc.
- fixpoint until stabilization
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    } else
        success;
}

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths
### Binary-level semantic approaches

#### Path predicate computation

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>input(y,z)</code></td>
</tr>
<tr>
<td>1</td>
<td><code>w := y+1</code></td>
</tr>
<tr>
<td>2</td>
<td><code>x := w + 3</code></td>
</tr>
<tr>
<td>3</td>
<td><code>if (x &lt; 2 * z) (branche True)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>if (x &lt; z) (branche False)</code></td>
</tr>
</tbody>
</table>
### Path predicate computation

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>input(y,z)</code></td>
</tr>
<tr>
<td>1</td>
<td><code>w := y + 1</code></td>
</tr>
<tr>
<td>2</td>
<td><code>x := w + 3</code></td>
</tr>
<tr>
<td>3</td>
<td><code>if (x &lt; 2 * z) (branche True)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>if (x &lt; z) (branche False)</code></td>
</tr>
</tbody>
</table>

Let $W_1 \triangleq Y_0 + 1$ in
### Binary-level semantic approaches

#### Path predicate computation

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) (branche True)</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) (branche False)</td>
</tr>
</tbody>
</table>

let \( W_1 \triangleq Y_0 + 1 \) in
let \( X_2 \triangleq W_1 + 3 \) in
### Binary-level semantic approaches

#### Path predicate computation

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input((y,z))</td>
</tr>
<tr>
<td>1</td>
<td>(w := y + 1)</td>
</tr>
<tr>
<td>2</td>
<td>(x := w + 3)</td>
</tr>
<tr>
<td>3</td>
<td>if ((x &lt; 2 \times z)) (branche True)</td>
</tr>
<tr>
<td>4</td>
<td>if ((x &lt; z)) (branche False)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{let } W_1 & \triangleq Y_0 + 1 \\
\text{let } X_2 & \triangleq W_1 + 3 \\
X_2 & < 2 \times Z_0
\end{align*}
\]
### Binary-level semantic approaches

#### Path predicate computation

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>input(y, z)</code></td>
</tr>
<tr>
<td>1</td>
<td><code>w := y + 1</code></td>
</tr>
<tr>
<td>2</td>
<td><code>x := w + 3</code></td>
</tr>
<tr>
<td>3</td>
<td><code>if (x &lt; 2 * z) (branche True)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>if (x &lt; z) (branche False)</code></td>
</tr>
</tbody>
</table>

Let \( W_1 \triangleq Y_0 + 1 \) in

Let \( X_2 \triangleq W_1 + 3 \) in

\[ X_2 < 2 \times Z_0 \land X_2 \geq Z_0 \]
Binary-level semantic approaches

Semantic exploration (2)

Crackme challenges

- input == secret $\iff$ success
- input $\neq$ secret $\iff$ failure
With BINSEC [https://youtu.be/0xUc2jbpjQo]

- find the path leading to success
- “invert” the conditions, find the secret: bunny_slope@flare.com
- check: it works!
Binary-level semantic approaches

Semantic exploration (2)

Applications
- coverage: solve(\(PC\))
- bug finding: solve(\(PC \land Error\))
- exploit finding: solve(\(PC \land Error \land Hijack \land Payload\))

- “invert” the conditions, find the secret: bunny_slope@flare.com
- check: it works!
Binary-level semantic approaches

Semantic exploration: keys

**Symbolic Execution**
- path predicate computation
- formula preprocessing + SMT solver
- sound execution of the program [path necessarily feasible]

![Symbolic Execution Diagram](image)

**Dynamic Symbolic Execution [DSE]**
- combine dynamic and symbolic reasoning
- much more robust [missing code, self-modification, etc.]
Platform Outline

- Preambule
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- Conclusion
The BINSEC Platform [CEA & Uni. Grenoble Alpes]

- open source, lgpl v2.1
- mostly OCaml, 30 kloc (and pintool in C++)
- dual goal
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers
- allows for combination of techniques
- current: multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerabilities and deobfuscation
- still young: beta-version just released [http://binsec.gforge.inria.fr/]

Thx to a bunch of enthusiastic students: Robin David, Adel Djoudi, Josselin Feist, Than Dihn Ta, Benjamin Farinier
Platform

BINSEC platform (2)

Simulation:
- Flat, regions, low-level regions semantics
- Dynamic disassembly

Static analysis:
- Generic fixpoint computation
- Interleaved CFG recovery (closed/degraded mode)
Platform
BINSEC platform (2)

- loader ELF/PE
- decoder (x86) + IR simplification
- 460/500 instructions: 380/380 “basic”, 80/120 SIMD, no float/system
- prefixes: op size, addr size, repetition
- standard syntactic disassembly techniques: recursive, linear, combination

Basic services to build analysis on:

- Simulation
- Static analysis [semantic disassembly] [Adel Djoudi – tacas15, sub. fm16]
- Symbolic execution [semantic exploration] [Robin David – saner16,issta16]
- Combinations
Platform
BINSEC platform (2)

Static analysis
- Generic fixpoint computation
- Safe CFG recovery
- Tradeoffs for correctness, precision, efficiency

Symbolic execution
- Path predicate optimization
- Generic concretization & symbolization
- Generic path search
- Pintool

Disassembler
DBA Stub
Dynamic disassembly
Static analysis:

Decoder + instruction-level and block-level simplification

01101
01001
10100
DBA simplifications

- **Instruction level simplifications**
  - Idiom simplifications [local rewriting rules]

- **Block level simplifications**
  - Constants propagation
  - Remove redundant assigns

- **Program level simplifications**
  - Flag slicing (remove must-killed variables)
  - granularity: function level + automatic summary of callees
Platform

DBA simplifications

- **Instruction level simplifications**
  - Idiom simplifications [local rewriting rules]

- **Block level simplifications**
  - Constants propagation
  - Remove redundant assigns

- **Program level simplifications**
  - Flag slicing (remove must-killed variables)
  - Granularity: function level + automatic summary of callees

**Approach**

- Inspired from standard compiler optim
- Targets: flags & temp
- Sound: w.r.t. incomplete CFG
- Inter-procedural (summaries)
# Platform

## DBA simplifications: Experiments

### Table: Program Performance Comparison

<table>
<thead>
<tr>
<th>Program</th>
<th>Native Loc</th>
<th>DBA Loc</th>
<th>Opt (DBA) Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>166K</td>
<td>559K</td>
<td>673.61s 389K 30.45%</td>
</tr>
<tr>
<td>cat</td>
<td>8K</td>
<td>23K</td>
<td>18.54s 18K 23.02%</td>
</tr>
<tr>
<td>echo</td>
<td>4K</td>
<td>10K</td>
<td>6.96s 8K 24.26%</td>
</tr>
<tr>
<td>less</td>
<td>23K</td>
<td>80K</td>
<td>69.99s 55K 30.96%</td>
</tr>
<tr>
<td>ls</td>
<td>19K</td>
<td>63K</td>
<td>65.69s 44K 30.58%</td>
</tr>
<tr>
<td>mkdir</td>
<td>8K</td>
<td>24K</td>
<td>19.74s 17K 29.50%</td>
</tr>
<tr>
<td>netstat</td>
<td>17K</td>
<td>50K</td>
<td>52.59s 40K 20.05%</td>
</tr>
<tr>
<td>ps</td>
<td>12K</td>
<td>36K</td>
<td>36.99s 27K 23.98%</td>
</tr>
<tr>
<td>pwd</td>
<td>4K</td>
<td>11K</td>
<td>7.69s 9K 23.56%</td>
</tr>
<tr>
<td>rm</td>
<td>10K</td>
<td>30K</td>
<td>24.93s 22K 25.24%</td>
</tr>
<tr>
<td>sed</td>
<td>10K</td>
<td>32K</td>
<td>28.85s 23K 26.20%</td>
</tr>
<tr>
<td>tar</td>
<td>64K</td>
<td>213K</td>
<td>242.96s 154K 27.48%</td>
</tr>
<tr>
<td>touch</td>
<td>8K</td>
<td>26K</td>
<td>24.28s 18K 27.88%</td>
</tr>
<tr>
<td>uname</td>
<td>3K</td>
<td>10K</td>
<td>6.99s 8K 23.62%</td>
</tr>
</tbody>
</table>

### Reduction

<table>
<thead>
<tr>
<th>Reduction</th>
<th>time</th>
<th>dba instr</th>
<th>tmp assigns</th>
<th>flag assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINSEC</td>
<td>1279.81s</td>
<td>28.64%</td>
<td>90.00%</td>
<td>67.04%</td>
</tr>
</tbody>
</table>
What can be reused?

- **whole analyses**
  - semantic exploration
  - semantic disassembly

- **basic blocks [need cleaner APIs]**
  - decoding
  - disassembly (cfg, call graph)
  - abstract fixpoint computation
  - path predicate, formula simplification & solving
  - generic path exploration
  - pintool
Achievements

Outline

- Preambule
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- Conclusion
Achievements

Finding *use-after-free* vulnerabilities

A pragmatic two-step approach implemented within the BINSEC plateform:
- not complete, but scalable and correct in some cases

- **GUEB**: scalable lightweight static analysis (not sound, not complete)
  → produces a set of CFGs slices containing *potential* UaF

- **BINSEC/SE**: guided symbolic execution
  → *confirm* the UaF by finding *concrete* program inputs
Help to find the needle in the heap
Achievements

Help to find the needle in the heap
Combination of techniques is fruitful!

Several **new** vulnerabilities found

- **GUEB + manual analysis** [j. comp. virology 14]
  - `tiff2pdf`: CVE-2013-4232
  - `openjpeg`: CVE-2015-8871
  - `gifcolor`: CVE-2016-3177
  - `accel-ppp`

- **GUEB + BINSE/SE** [sefm16]
  - `Jasper JPEG-2000`: CVE-2015-5221

Gueb [Josselin Feist]
- MIT licence
- Ocaml, 5kloc
- https://github.com/montyly/gueb
Achievements
Malware deobfuscation

Context: x86-malware

A common protection scheme for malware
a SillyFDC run

Self-modifying program schema
Achievements

Malware deobfuscation

Context: x86-malware

BINSEC/SE [saner16, sub. ccs16]
- malware exploration (vxheaven)
- detection of opaque predicates (o-llvm)
- detection of stack tampering (tigress)
- experiments on commercial packers

- static analysis: not safe, complete, not robust to obfuscation
- dynamic analysis: safe, not complete, robust to obfuscation
- symbolic execution: best of both world
- fruitful combination dynamic, static, symbolic

Self-modifying program schema
Conclusion

Outline

- Preambule
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- At last
Conclusion

Binary-level security analysis
- many applications, many challenges
- syntactic and dynamic are not enough

Semantic approaches can help!
- semantic exploration, semantic disassembly
- yet, still hard to design

The BINSEC Platform [CEA & Uni. Grenoble Alpes]
- open source, dual goal
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers [already a few ones]
- current: multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerability detection and deobfuscation
- still young: beta-version just released [http://binsec.gforge.inria.fr/]

BINSEC team RMLL 2016: The Security Track
Conclusion

**Binary-level security analysis**

- many applications, many challenges
- syntactic and dynamic are not enough

**In progress**

- tutorials, doc
- code cleaning
- ARM and PowerPC

**The BINSEC Platform** [CEA & Uni. Grenoble Alpes]

- open source, dual goal
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers [already a few ones]
- current: multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerability detection and deobfuscation
- still young: beta-version just released [http://binsec.gforge.inria.fr/]
**Binary-level security analysis**

- many applications, many challenges
- syntactic and dynamic are not enough
- semantic approaches can help!
  - semantic exploration, semantic disassembly
  - yet, still hard to design

**The BINSEC Platform** [CEA & Uni. Grenoble Alpes]

- open source, dual goal
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers (already a few ones)
- current: multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerability detection and deobfuscation
- still young: beta-version just released [http://binsec.gforge.inria.fr/]

**In progress**

- tutorials, doc
- code cleaning
- ARM and PowerPC

**Formal methods for software analysis**

- lots of effort in proprietary industry
- open source community needs to keep up the pace