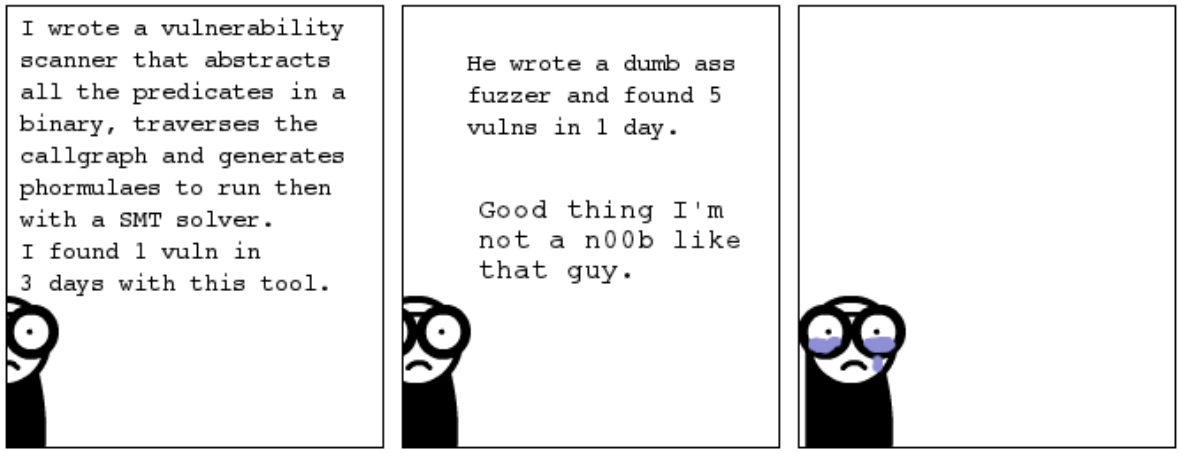
Modern static security checking of C / C++ programs

RECON: Reverse Engineering Conference
Montreal, June 14th 2012

Julien Vanegue - Microsoft Security Science team
Shuvendu K. Lahiri - Microsoft Research



Trustworthy Computing – 10 years

Protecting Microsoft customers across the lifecycle
(in development, deployment & operations)

Microsoft Security Response Center (MSRC)	Microsoft Security Engineering Center (MSEC)	Microsoft Malware Protection Center (MMPC)	Network Security (NETSEC)

2002 - 2003

- Bill Gates writes "Trustworthy Computing" memo early 2002
- "Windows security push" for Windows Server 2003
- Security push and FSR extended to other products

2004

- Microsoft Senior Leadership Team agrees to require SDL for all products that:
  - Are exposed to meaningful risk and/or
  - Process sensitive data

2005 - 2007

- SDL is enhanced
  - "Fuzz" testing
  - Code analysis
  - Crypto design requirements
  - Privacy
  - …
- Windows Vista is the first OS to go through full SDL cycle

Now

- Optimize the process through feedback, analysis and automation
- Evangelize the SDL to the software development community

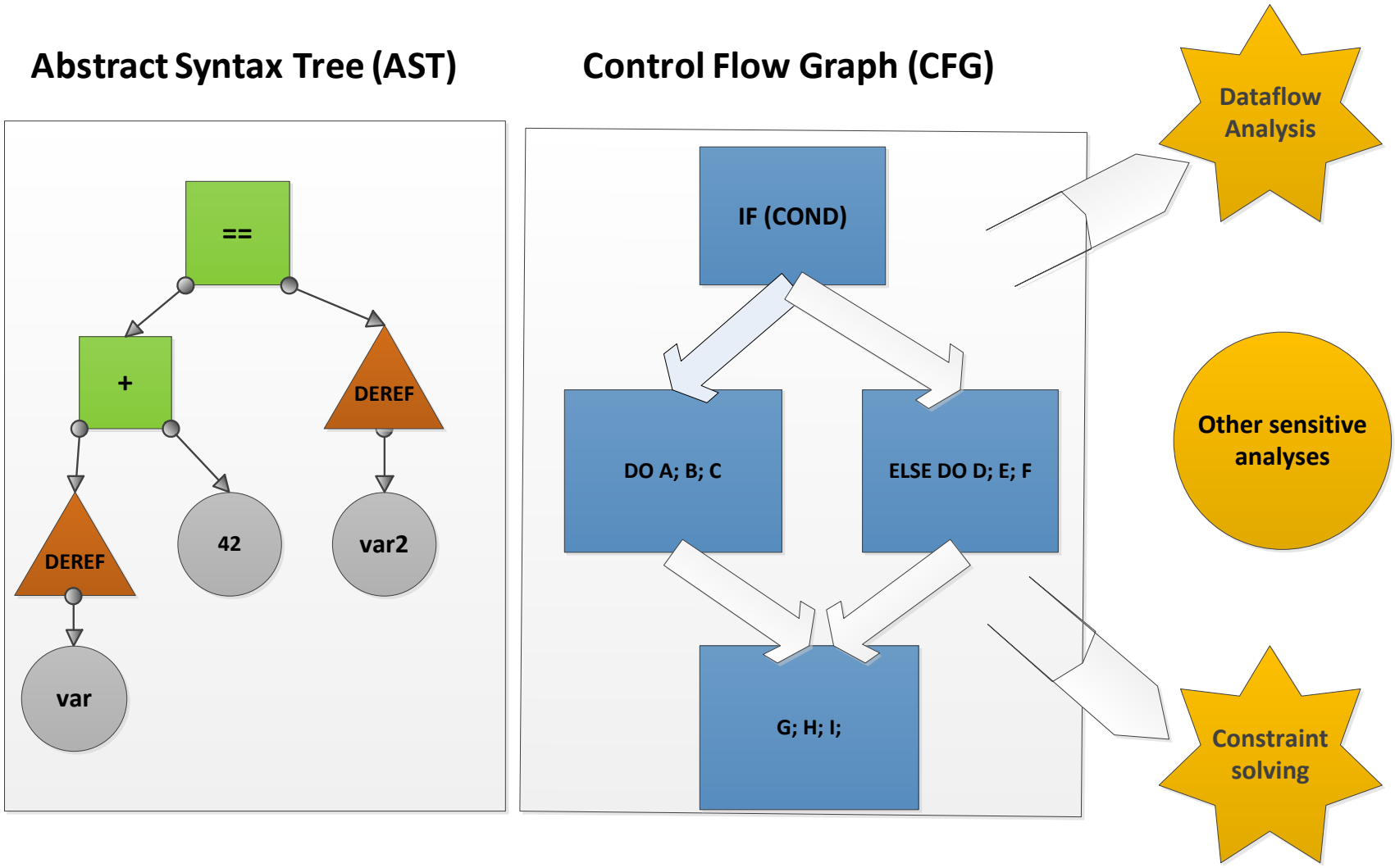
Covered today

- General introduction to static analysis
- Incubating Microsoft tools to perform static security analysis at the source level (C/C++ programs)
  - Interesting safety properties of pointers, arrays and structures that can be checked with a reasonable signal/noise ratio.
  - Tackling sequential properties for today (no checking for concurrency vulnerabilities).
  - Checking loop invariants, function pre/post conditions, and deep inter-procedural analysis with HAVOC/Boogie/Z3.

- NOT covered: data-flow analysis tools based on state merging algorithm in the Microsoft compiler Esp framework.

Part 1: Introduction to static analysis

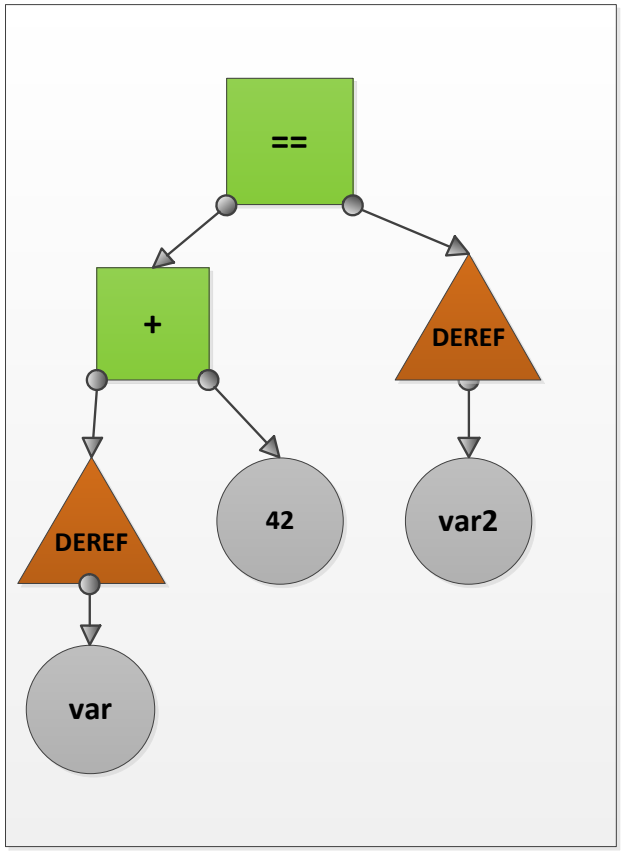
The program analysis spectrum



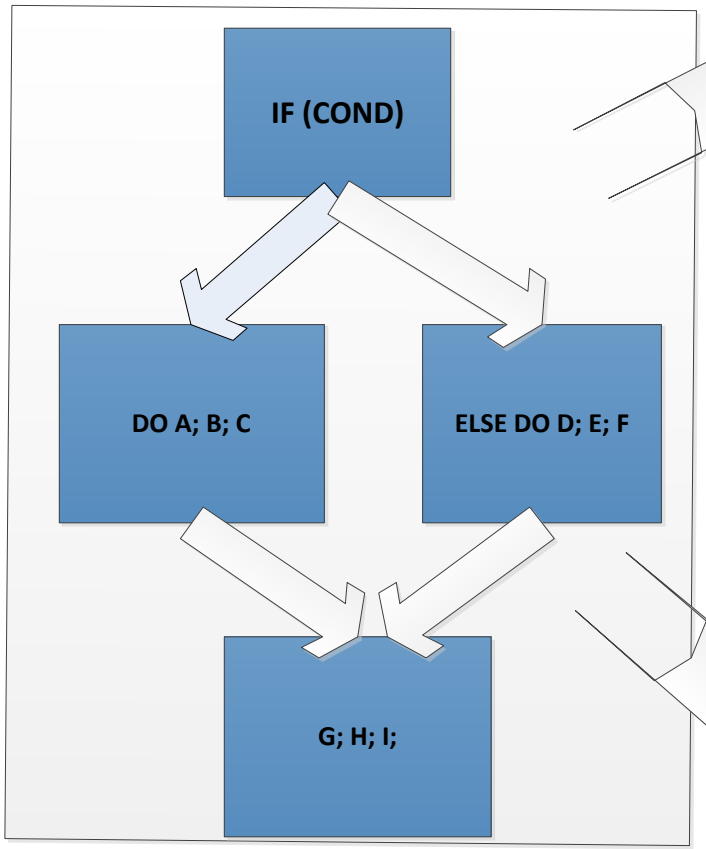
To check a given property, do you need flow sensitivity? Path sensitivity? Context sensitivity?

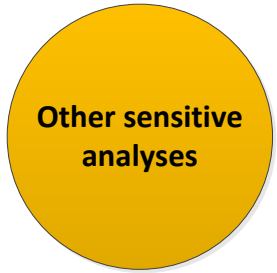
The program analysis spectrum

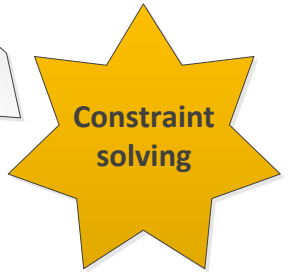
Abstract Syntax Tree (AST)



Control Flow Graph (CFG)

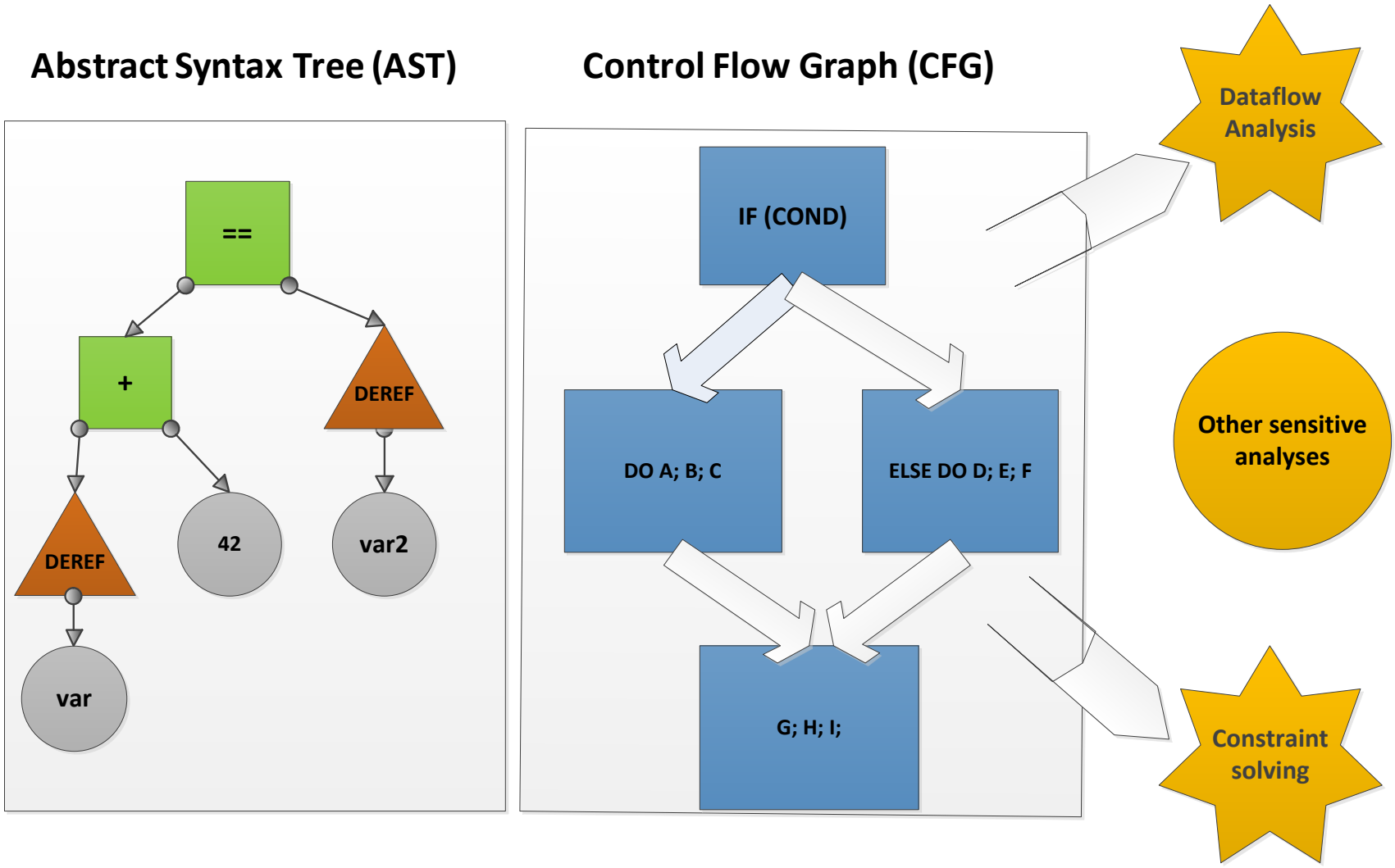






Question to audience: where is placed type checking for C programs ?

The program analysis spectrum



Answer: Just after AST transformation – type checking for C is NOT flow sensitive

Sound vs. Complete program analysis

Sound (no false negative)

Complete (no false positive)

A consequence of Gödel's incompleteness theorem is that there exists some (classes of) programs for which it is not possible to prove important properties (such as termination – See the Turing machine halting problem).

We often need a trade-off between soundness and completeness

Sound vs. Complete program analysis

Sound (no false negative)

Complete (no false positive)

A consequence of Gödel's incompleteness theorem is that there exists some (classes of) programs for which it is not possible to prove important properties (such as termination – See the Turing machine halting problem).

Question to audience: where do you place fuzz testing?

Sound vs. Complete program analysis

Sound (no false negative)

Complete (no false positive)

FUZZING is UNSOUND (SOME FALSE NEGATIVES)
and COMPLETE (NO FALSE POSITIVE)

Do not mistake *sound program analysis* (the ability to find all instances of a particular bug class without false negatives) and the *soundness of bugs* (the guarantee that static analysis warnings are real bugs, e.g. analysis *completeness*).

Dealing with false positives in practice

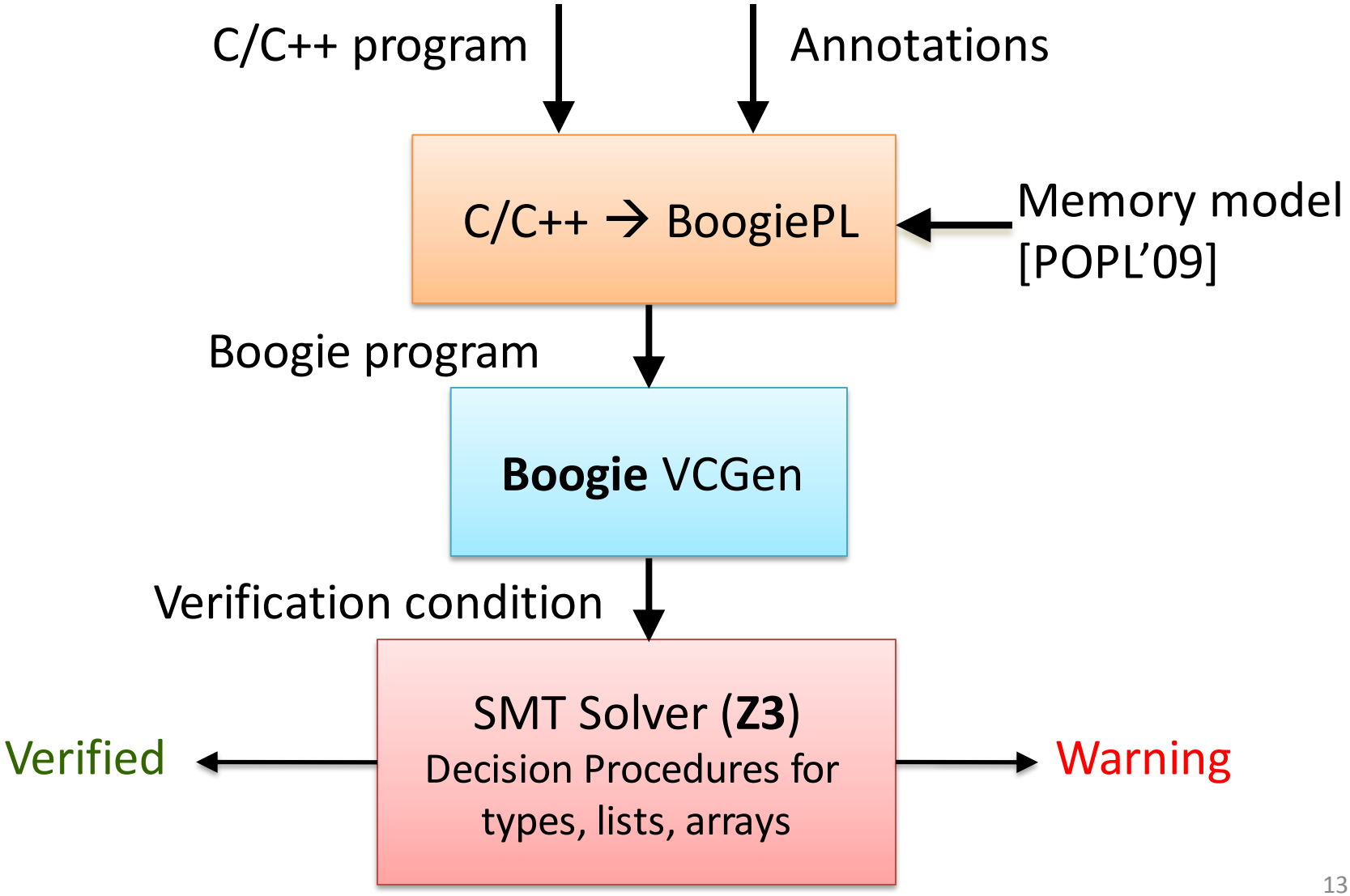
- **Manual refinement**: add manual annotations to inform the analyzer that it needs to track specific boundary conditions of specific variables at certain functions pre- or post-states.
- **Warning prioritization:** give a level of confidence to static analysis alerts based on pre-selected criteria to ensure that best warnings make it to the top of the list.
- **Last resort: Lose soundness (add potential false negatives)**
  - Add assumptions that cannot be proved but greatly reduce the number of warnings (ex: say that specific macros are safe)
  - Pre-filter attack surface based on known API, type, or variable usage to focus on the most shallow bugs.

HAVOC: Heap Aware Verifier for C and C++ programs

- Developed at Microsoft Research in the RiSE team : : *http://research.microsoft.com/en-us/projects/havoc/*
- Plug-in for the Microsoft C/C++ compiler.
- Detailed user manual.
- Based on the (open source) Boogie theorem prover : *http://boogie.codeplex.com*
- Decision procedure based on constraint solver Z3
- User can specify properties to be checked via annotations.

The Microsoft Security team uses HAVOC to find new and variants of existing vulnerabilities (“*variation hunting*”)

HAVOC: Heap aware verifier for C/C++ programs



HAVOC: main features

HAVOC users can make use of three main constructs:

- **Requires(X)** F(); check the validity of a pre-condition **X** at function/method **F** initial state.

- **Ensures(X)** F(); check the validity of a post-condition **X** at function/method **F** final state.

- **_resource["MAP_NAME", varname] == Y** provides a mechanism to track symbolic values *("ghost fields"*) for specific variables. Here the symbolic value **Y** is associated to variable **varname**. **MAP_NAME** is the name of the resource (you can have many). This is used inside requires() or ensures() to track symbolic values across function boundaries.

- **Requires** and **Ensures** can be prefixed by **__free_**, in that case they are assumed and not checked (more on this later).

Dummy example [3] (step 1)

```
int f(UINT val, int mode)
{
UINT size, pad = 0;
if (val > 8) return ERR;
size = val * 2;
if (mode == M32)
  pad = sizeof(T32);
else if (mode == M64)
   pad = sizeof(T64);
size += pad;

PTYPE ptr = Alloc(size);
```

Verification condition

f1: pad=0

__requires(size != 0)

Dummy example (step 2)

```
int f(UINT val, int mode)
{
UINT size, pad = 0;
if (val > 8) return ERR;
size = val * 2;
if (mode == M32)
  pad = sizeof(T32);
else if (mode == M64)
   pad = sizeof(T64);
size += pad;

PTYPE ptr = Alloc(size);
```

Verification condition

f1: pad=0

f2: f1 && (size == val*2) && (val <= 8)

__requires(size != 0)

Dummy example (step 3)

```
int f(UINT val, int mode)
{
UINT size, pad = 0;
if (val > 8) return ERR;
size = val * 2;
if (mode == M32)
  pad = sizeof(T32);
else if (mode == M64)
   pad = sizeof(T64);
size += pad;

PTYPE ptr = Alloc(size);
```

Verification condition

f1: pad=0

f2: f1 && (size == val*2) && (val <= 8)

f3: pad=sizeof(T32) && (size == val*2) && (val <= 8)

__requires(size != 0)

Dummy example (step 4)

```
int f(UINT val, int mode)
{
UINT size, pad = 0;
if (val > 8) return ERR;
size = val * 2;
if (mode == M32)
  pad = sizeof(T32);
else if (mode == M64)
   pad = sizeof(T64);
size += pad;

PTYPE ptr = Alloc(size);
```

Verification condition

f1: pad=0

f2: f1 && (size == val*2) && (val <= 8)

f3: pad=sizeof(T32) && (size == val*2) && (val <= 8)

f4: pad=sizeof(T64) && (size == val*2) && (val <= 8)

__requires(size != 0)

Dummy example (step 5)

```
int f(UINT val, int mode)
{
UINT size, pad = 0;
if (val > 8) return ERR;
size = val * 2;
if (mode == M32)
  pad = sizeof(T32);
else if (mode == M64)
   pad = sizeof(T64);
size += pad;

PTYPE ptr = Alloc(size);
```

Verification condition

f1: pad=0

f2: f1 && (size == val*2) && (val <= 8)

f3: pad=sizeof(T32) && (size == val*2) && (val <= 8)

f4: pad=sizeof(T64) && (size == val*2) && (val <= 8)

f5: (size == val*2 + pad) && (val<=8) &&
(pad=0 || pad=sizeof(T32) || pad=sizeof(T64))

__requires(size != 0)

Dummy example (step 6)

```
int f(UINT val, int mode)
{
UINT size, pad = 0;
if (val > 8) return ERR;
size = val * 2;
if (mode == M32)
  pad = sizeof(T32);
else if (mode == M64)
   pad = sizeof(T64);
size += pad;

PTYPE ptr = Alloc(size);
```

Verification condition

f1: pad=0

f2: f1 && (size == val*2) && (val <= 8)

f3: pad=sizeof(T32) && (size == val*2) && (val <= 8)

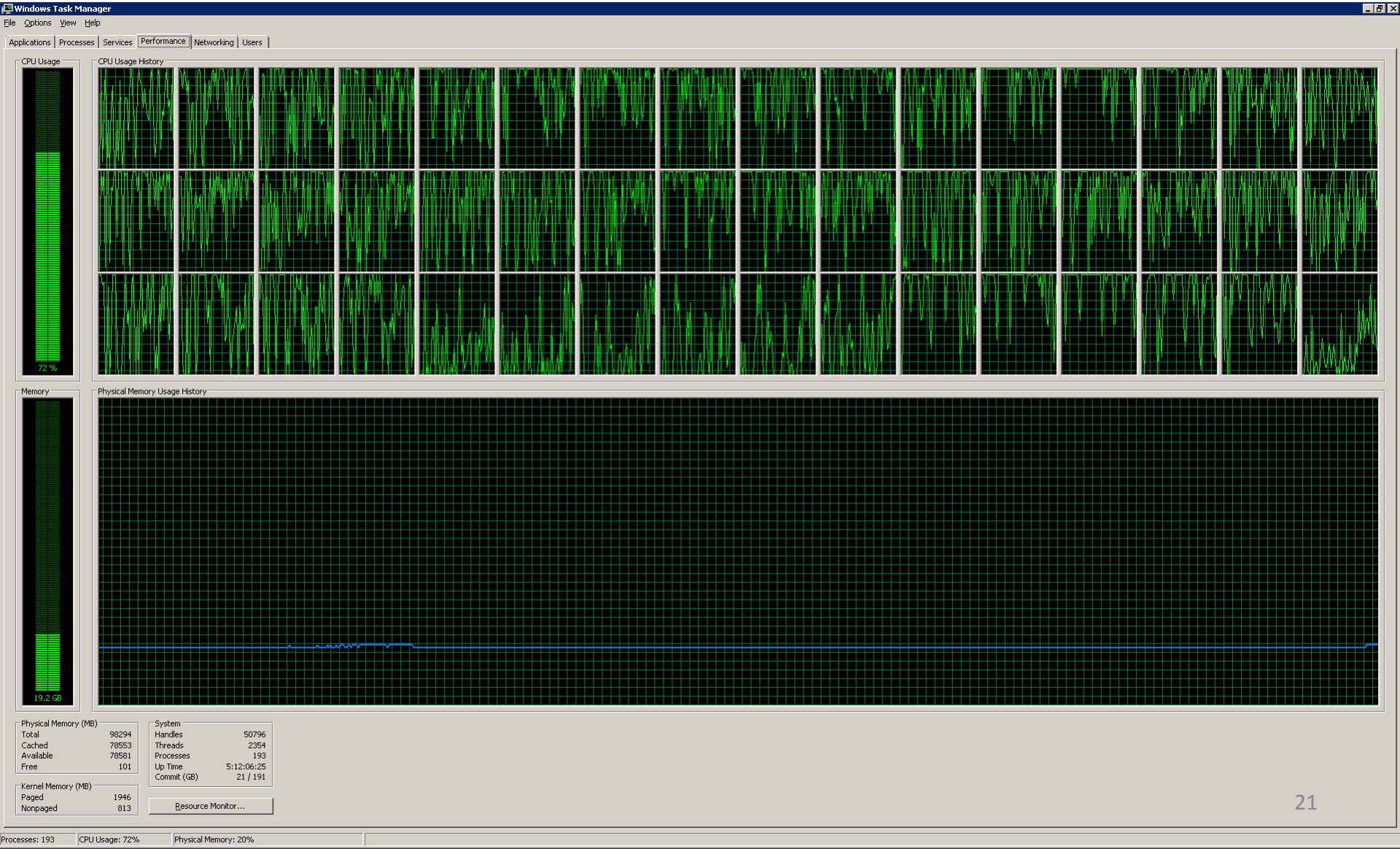
f4: pad=sizeof(T64) && (size == val*2) && (val <= 8)

f5: (size == val*2 + pad) && (val<=8) &&
(pad=0 || pad=sizeof(T32) || pad=sizeof(T64))

__requires(size != 0) ← Precondition violation!

HAVOC retains path-sensitivity at merge points without approximation
(See BONUS SLIDE with details on single assignments and variable versioning)

Let the party begin..



Part 2: OO awareness, checking loops, deep inter-procedural analysis examples using HAVOC.

Ex 1: Webkit CSS type confusion information disclosure vulnerability

- CVE-2010-4577: *"Google Chrome before 8.0.552.224 and Chrome OS before 8.0.552.343 do not properly parse Cascading Style Sheets (CSS) token sequences, which allows remote attackers to read stack content"*
- Published and exploited by Chris Rohlf [7]
- Illustrative example used by Sean Heelan [5] to show where static analysis could be useful for code security.
- We show how to analyze those vulnerabilities using HAVOC/Boogie/Z3.

Simplified version of the webkit bug

```
bool CSSParser::parseFontFaceSrc()
{
    CSSValueList values(CSSValueList::createCommaSeparated());
    CSSParserValue* val;
    while ((val = m_valueList->current())) {
        CSSFontFaceSrcValue *parsedValue = NULL;
         if (val->unit == CSSParserValue::Function) {
           CSSParserValueList* args = val->function->args;
           if (args && args->size() == 1) {
              if (equalIgnoringCase(val->function->name, "local(")
+                 && (args->current()->unit == CSSPrimitiveValue::CSS_STRING ||
+                args->current()->unit == CSSPrimitiveValue::CSS_IDENT)) {
                CSSParserValue* a = args->current();
               // bug if variable a is NOT of string type! Fix: uncomment green lines
                parsedValue = CSSFontFaceSrcValue::createLocal(a->string);
            } } }
        if (parsedValue) values.append(parsedValue->release());
        m_valueList->next();
    }
    return false;
}
```

Check the example using HAVOC static instrumentation capabilities

```
// Mandate that unit field has value STRING or IDENT before using the address of the
string field in a CSSParserValue structure (e.g. &v->string)
__requires(v->unit == CSS_STRING || v->unit == CSS_IDENT)
__instrument_address_pre(v->string)
void __instrument_access_hook(CSSParserValue *v){ return; }

// Same for write to the string field (e.g. v->string = val)
__requires(v->unit == CSS_STRING || v->unit == CSS_IDENT)
__instrument_write_pre(v->string)
void __instrument_write_hook(CSSParserValue *v){ return; }

// Same for read from the string field (e.g. val = v->string)
__requires(v->unit == CSS_STRING || v->unit == CSS_IDENT)
__instrument_read_pre(v->string)
void __instrument_read_hook(CSSParserValue *v){ return; }
```

Directives are written in a side file, no annotation is needed in the analyzed code.

Checking example 1(demo)

```
$ Boogie.exe  parseFontFaceSrc$CSSParser.bpl
parseFontFaceSrc$CSSParser.bpl(1045,1): Error BP5002: A precondition for this call might not hold.
parseFontFaceSrc$CSSParser.bpl(576,1): Related location: This is the precondition that might not hold. Execution trace:
  parseFontFaceSrc$CSSParser.bpl(823,1): start
  parseFontFaceSrc$CSSParser.bpl(837,1): label_7
  parseFontFaceSrc$CSSParser.bpl(843,1): label_10
  parseFontFaceSrc$CSSParser.bpl(849,1): label_4
  parseFontFaceSrc$CSSParser.bpl(860,1): label_12
  parseFontFaceSrc$CSSParser.bpl(972,1): label_25_true
  parseFontFaceSrc$CSSParser.bpl(983,1): label_26
  parseFontFaceSrc$CSSParser.bpl(993,1): label_29_true
  parseFontFaceSrc$CSSParser.bpl(1004,1): label_30
  parseFontFaceSrc$CSSParser.bpl(1016,1): label_33_true
  parseFontFaceSrc$CSSParser.bpl(1032,1): label_35
  parseFontFaceSrc$CSSParser.bpl(1044,1): label_39
$
```

- We run Boogie on the vulnerable code: pre-condition is found to be violated (the bug is found). Good!

- We uncomment the fix line: **problem**: the code location is still marked as vulnerable! Why?
Reason: There is no guarantee that args->current() returns the same value at every call, we have to explain this to the analyzer.

- The below annotation makes the false positive disappear in the fixed version (needed to explain that the result of method current() only depends on its parameters (the *this* pointer)

Refinement post-condition:

```
__ensures(__return == __resource("CUR_FROM_ARGS", this))
CSSParserValue* CSSParserValueList::current();
```

Lesson learned from example 1

- When vulnerability classes are generic, instrumentations can be used to make the contract explicit without pre-existing annotations.
- HAVOC is sound in that it will have false positives but no false negatives (unless initial assumptions are unsound).
- Manual annotations can be used to craft a very polished version of the checker. However those are not mandatory when using a tool as an aid to code review (unless signal/noise ratio is too low – heavily depends on the property being checked)

Example 2
Deep inter-procedural analysis

"Every pointer entering the OS kernel via one of the entry points is validated before being dereferenced"

→ Applied to large core kernel components of Windows (300KLOC)

Automated analysis workflow (3 steps)

1. A pre-analysis looks at the types of parameters for all functions and generates a candidate invariant **candrequires(checked(ptr))**

2. The Houdini algorithm [2] runs on the call graph and decides which candidates hold in all possible function contexts (else, the candidate is removed). Remains all proved candidates.

3. Every pointer variable are proved to be checked before dereferenced (assuming initial function conditions proved at second step). This step is intra-procedural only.

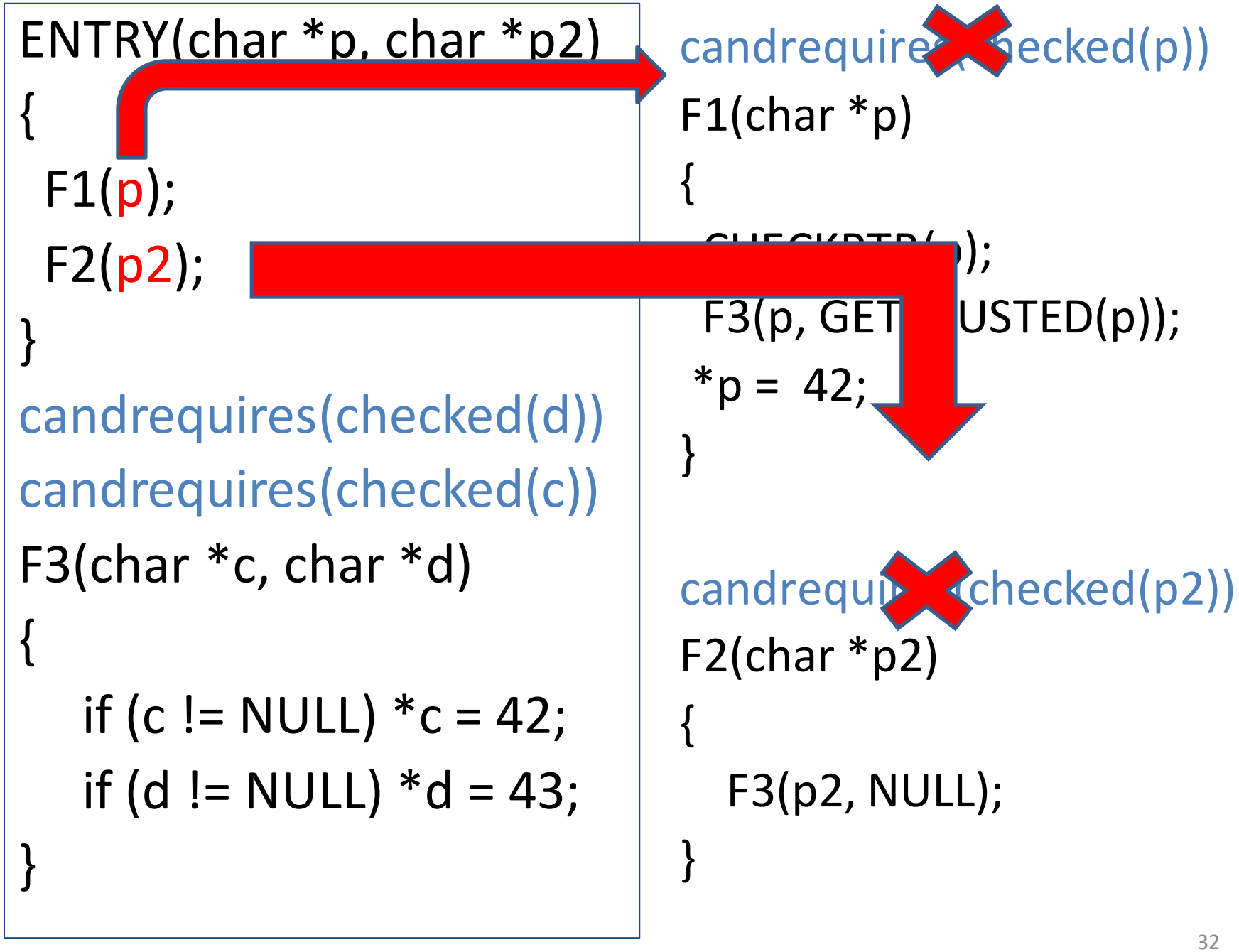
This analysis can be completely automated because the candidate contracts are very simple and well identified as "checked(ptr)"

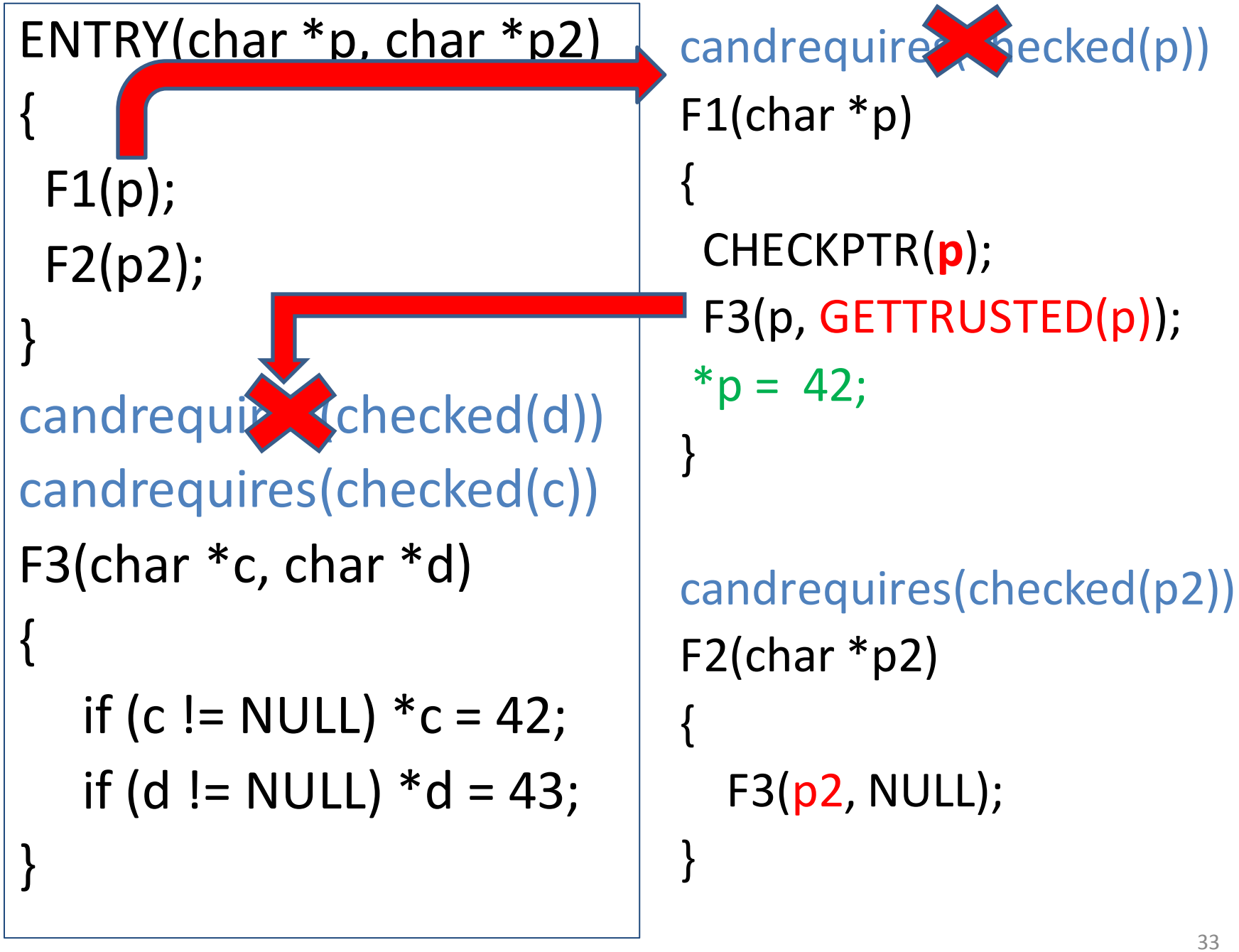
```
ENTRY(char *p, char *p2)
{
  F1(p);
  F2(p2);
}
F3(char *c, char *d)
{
    if (c != NULL) *c = 42;
    if (d != NULL) *d = 43;
}
```

```
F1(char *p)
{
  CHECKPTR(p);
  F3(p, GETTRUSTED(p));
 *p =  42;
}

F2(char *p2)
{
    F3(p2, NULL);
}
```

```
ENTRY(char *p, char *p2)
{
  F1(p);
  F2(p2);
}
candrequires(checked(d))
candrequires(checked(c))
F3(char *c, char *d)
{
    if (c != NULL) *c = 42;
    if (d != NULL) *d = 43;
}
```

```
candrequires(checked(p))
F1(char *p)
{
  CHECKPTR(p);
  F3(p, GETTRUSTED(p));
 *p =  42;
}

candrequires(checked(p2))
F2(char *p2)
{
   F3(p2, NULL);
}
```


ENTRY(char *p, char *p2)
{
F1(p);
F2(p2);
}
candrequires(checked(d))
candrequires(checked(c))
F3(char *c, char *d)
{
if (c != NULL) *c = 42;
if (d != NULL) *d = 43;
}
candrequires(checked(p))
F1(char *p)
{
F3(p, GET
USTED(p));
*p = 42;
}
candrequi
checked(p2))
F2(char *p2)
{
F3(p2, NULL);
}

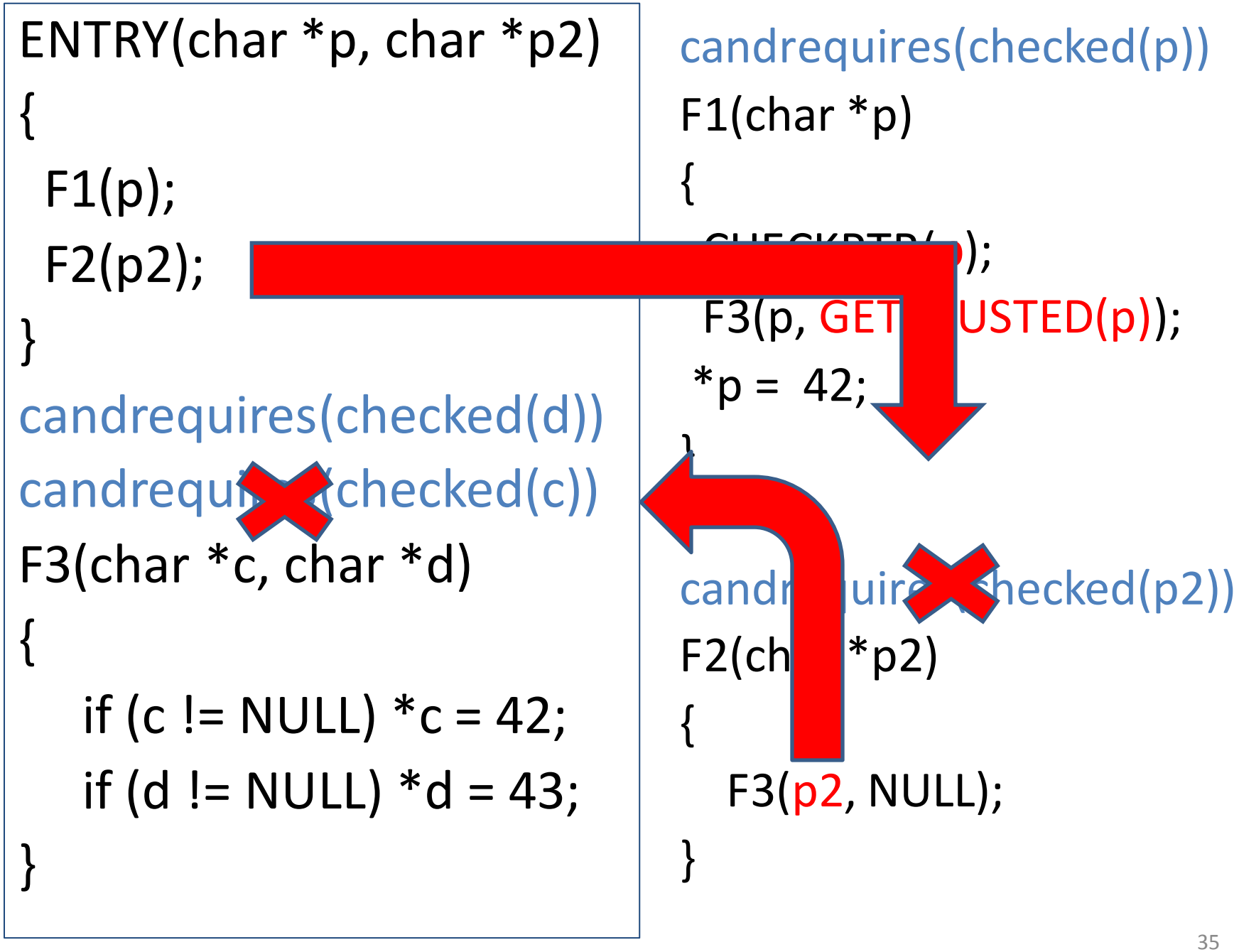

ENTRY(char *p, char *p2)
{
F1(p);
F2(p2);
}
candrequires(checked(d))
candrequires(checked(c))
F3(char *c, char *d)
{
if (c != NULL) *c = 42;
if (d != NULL) *d = 43;
}
candrequires(checked(p))
F1(char *p)
{
CHECKPTR(p);
F3(p, GETTRUSTED(p));
*p = 42;
}
candrequires(checked(p2))
F2(char *p2)
{
F3(p2, NULL);
}

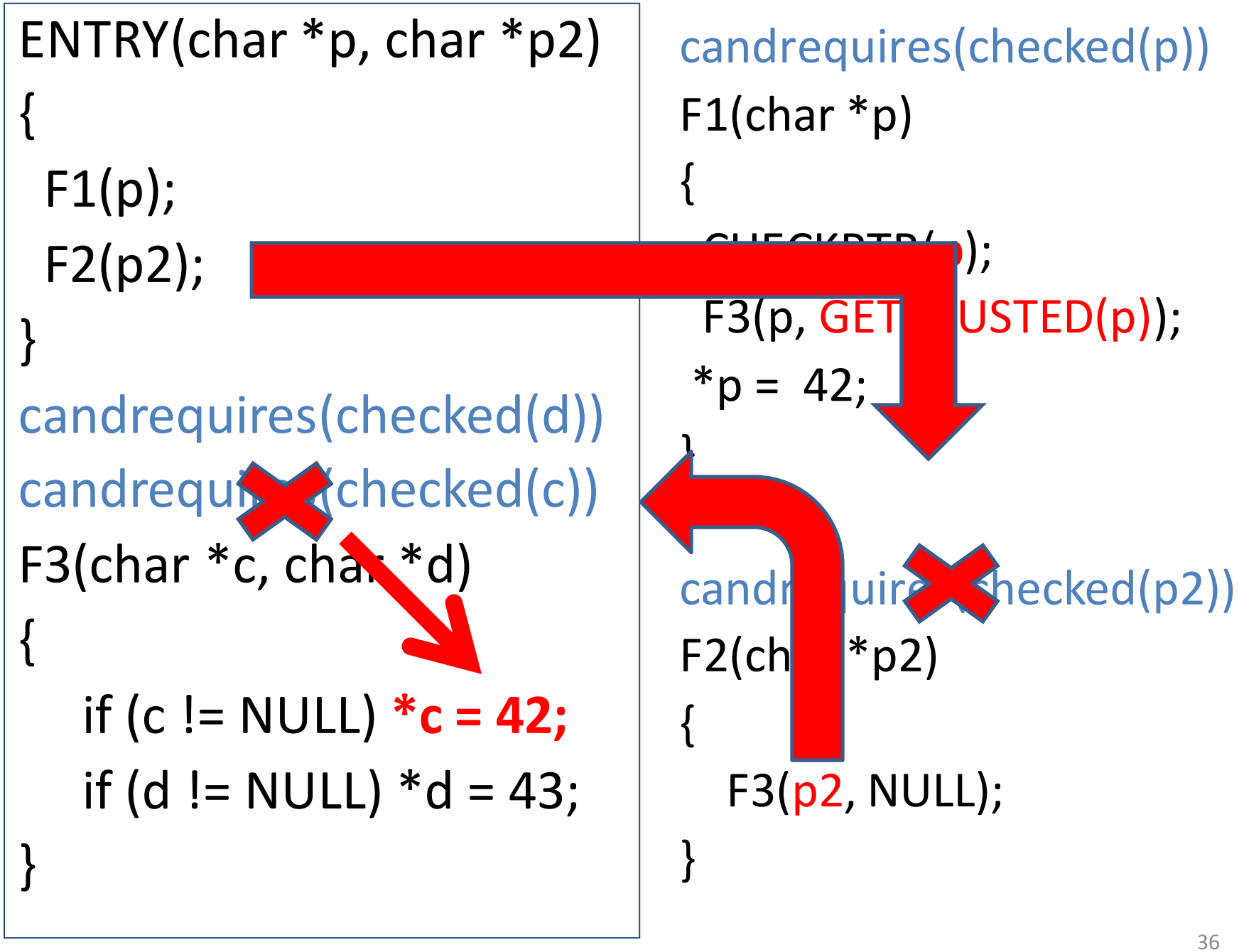
```
ENTRY(char *p, char *p2)
{
  F1(p);
  F2(p2);
}
candrequires(checked(d))
candrequires(checked(c))
F3(char *c, char *d)
{
    if (c != NULL) *c = 42;
    if (d != NULL) *d = 43;
}
```

```
candrequires(checked(p))
F1(char *p)
{
  CHECKPTR(p);
  F3(p, GETTRUSTED(p));
 *p =  42;
}
```

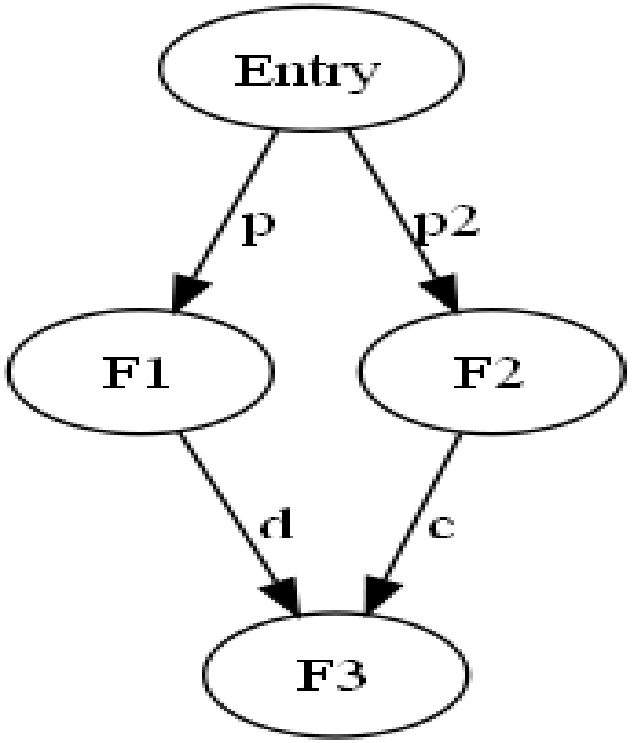
Needed post-condition
ensures(checked(__return))

```
candrequires(checked(p2))
F2(char *p2)
{
   F3(p2, NULL);
}
```


ENTRY(char *p, char *p2)
{
F1(p);
F2(p2);
}
candrequires(checked(d))
candrequires(checked(c))
F3(char *c, char *d)
{
if (c != NULL) *c = 42;
if (d != NULL) *d = 43;
}
candrequires(checked(p))
F1(char *p)
{
F3(p, GET USTED(p));
*p = 42;
}
candr uire checked(p2))
F2(ch *p2)
{
F3(p2, NULL);
}

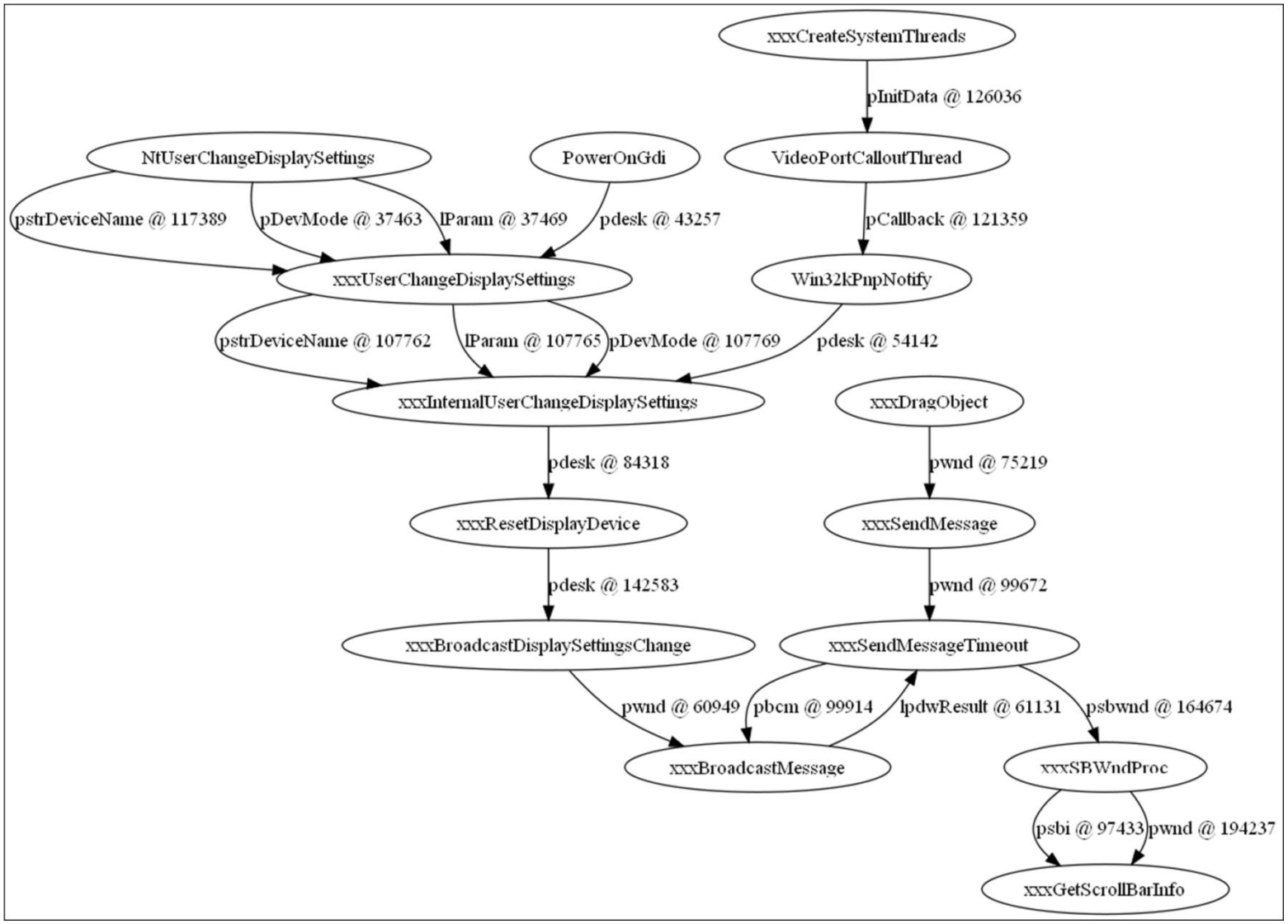

ENTRY(char *p, char *p2)
{
F1(p);
F2(p2);
}
candrequires(checked(d))
candrequires(checked(c))
F3(char *c, char *d)
{
if (c != NULL) *c = 42;
if (d != NULL) *d = 43;
}
candrequires(checked(p))
F1(char *p)
{
F3(p, GET
USTED(p));
*p = 42;
}
candr
uire
hecked(p2))
F2(ch
*p2)
{
F3(p2, NULL);
}

Inter-procedural inference graph

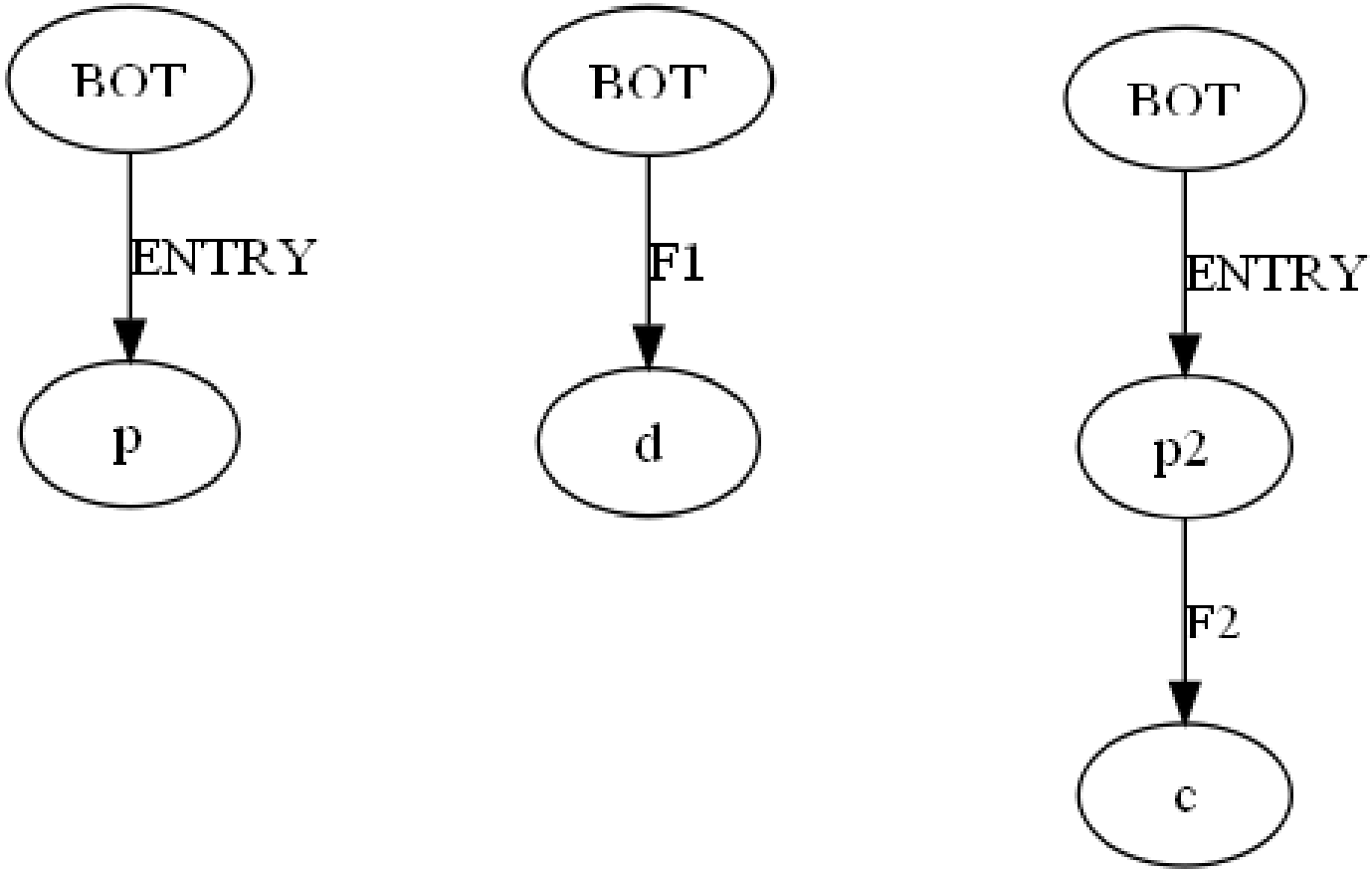


Causality of inference is hard to understand by looking at the raw output of Houdini.

Houdini graph for a real scenario



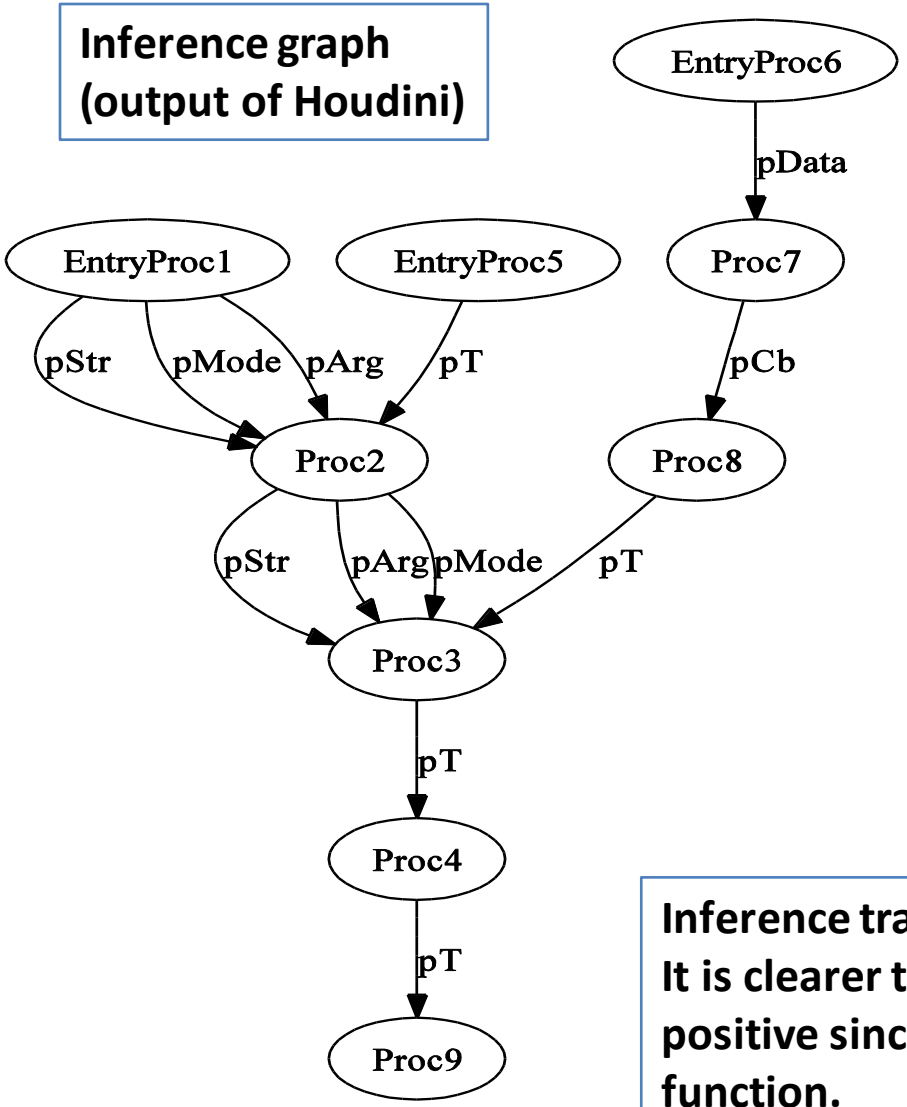
Refined causality traces

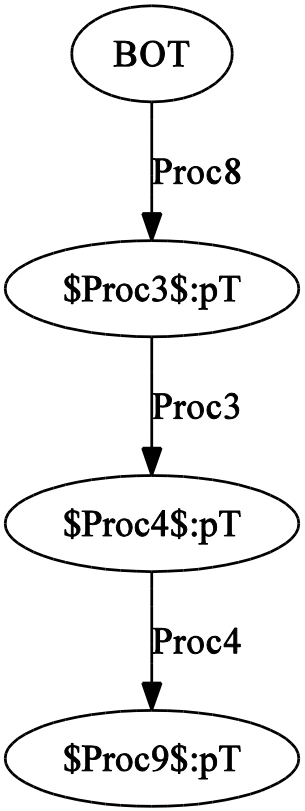


Those graphs are the real explanation of inference. We can obtain them with a very small modification of the Houdini algorithm (See ExplainHoudini [2]) .

ExplainHoudini typical usage : Filter out false positives

Inference graph (output of Houdini)





Inference trace (output of ExplainHoudini [2])
It is clearer that the problem is more likely a false positive since it is not rooted by an entry point function.

Example 3: Loop analysis
Sendmail CrackAddr() buffer overflow

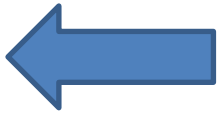
- CVE-2002-1337: "*A buffer overflow in sendmail 5.79 to 8.12.7 allows remote attackers to execute arbitrary code via certain formatted address fields, related to sender and recipient header comments as processed by the crackaddr function of headers.c*"
- Published by Mark Dowd [6] , Exploited by Last Stage of Delirium group in 4 hours (bugtraq posts).
- Presented at Infiltrate 2011 by Thomas Dullien [4] as an example of failure of static analysis tools based on state merging algorithms.
- We show how to check the absence of such vulnerabilities using loop invariants in Havoc/Boogie/Z3.

CrackAddr() detection - Disclaimer

- Dullien's challenge [4] is a toy example and does not entirely reflects the crackaddr() bug:
  - The original fix is bigger than one line of code and address the vulnerability at multiple locations in the loop.
  - The original loop has more than two states, some of which are not taken into account here.
  - We take Dullien's example unmodified to respect the challenge settings and keep it simple / pedagogical.
  - We have not tried the technique on the original full-blown example and assume that the loop invariant would need modification.
- Our solution is not entirely automated as the user needs to provide a loop invariant. Automatically generating such loop invariant in a generic way is a research problem.

ample_bad.c

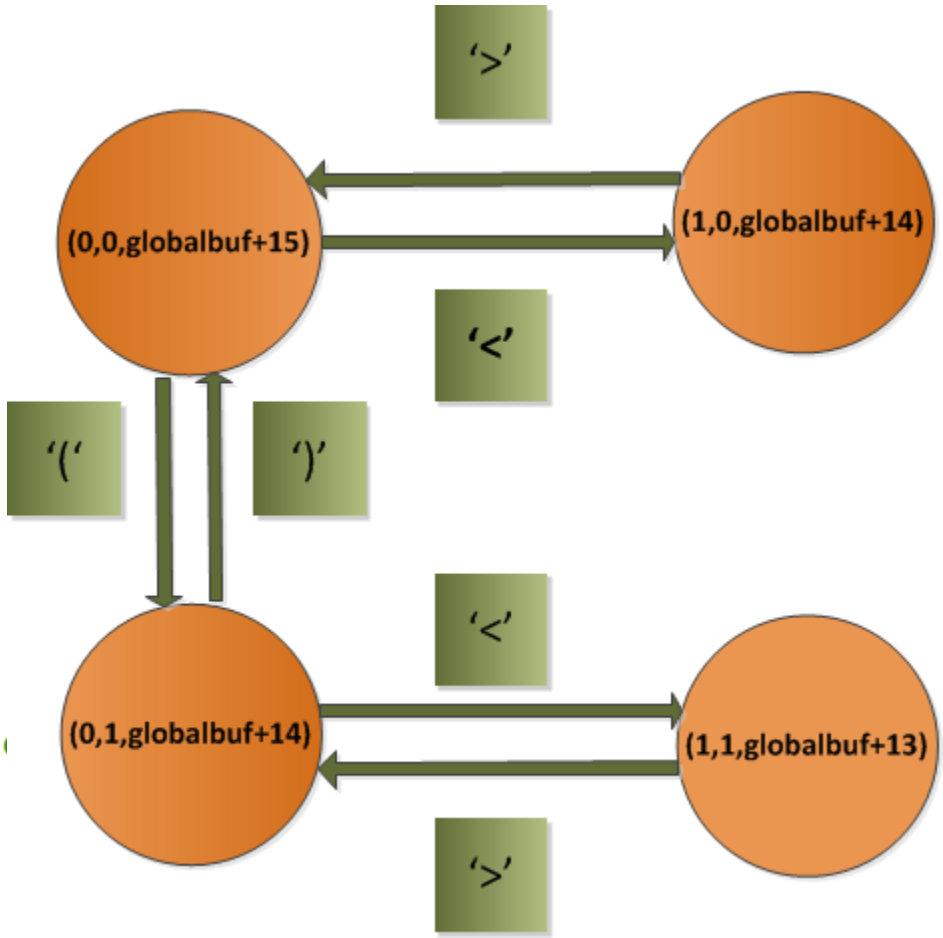
```
copy_it( char * input ){
  char localbuf[ BUFFERSIZE ];
  char c, *p = input, *d = &localbuf[0];
  char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
  int quotation = FALSE;
  int roundquote = FALSE;

  memset( localbuf, 0, BUFFERSIZE );
  while( (c = *p++) != '\0' ){
        if(( c == '<' ) && (!quotation)){
              quotation = TRUE;
              upperlimit--;}
        if(( c == '>' ) && (quotation)){
              quotation = FALSE;
              upperlimit++;}
        if(( c == '(' ) && ( !quotation ) && !roundquote){
              roundquote = TRUE;
              /*upperlimit--;*/}
        if(( c == ')' ) && ( !quotation ) && roundquote){
              roundquote = FALSE;
              upperlimit++;}
        // If there is sufficient space in the buffer, write the character.
        if( d < upperlimit )
              *d++ = c;
  }
  if( roundquote )
        *d++ = ')';
  if( quotation )
        *d++ = '>';

  printf("%d: %s\n", (int)strlen(localbuf), localbuf);
```



In our example,
BUFFERSIZE = 25

ample_bad.c

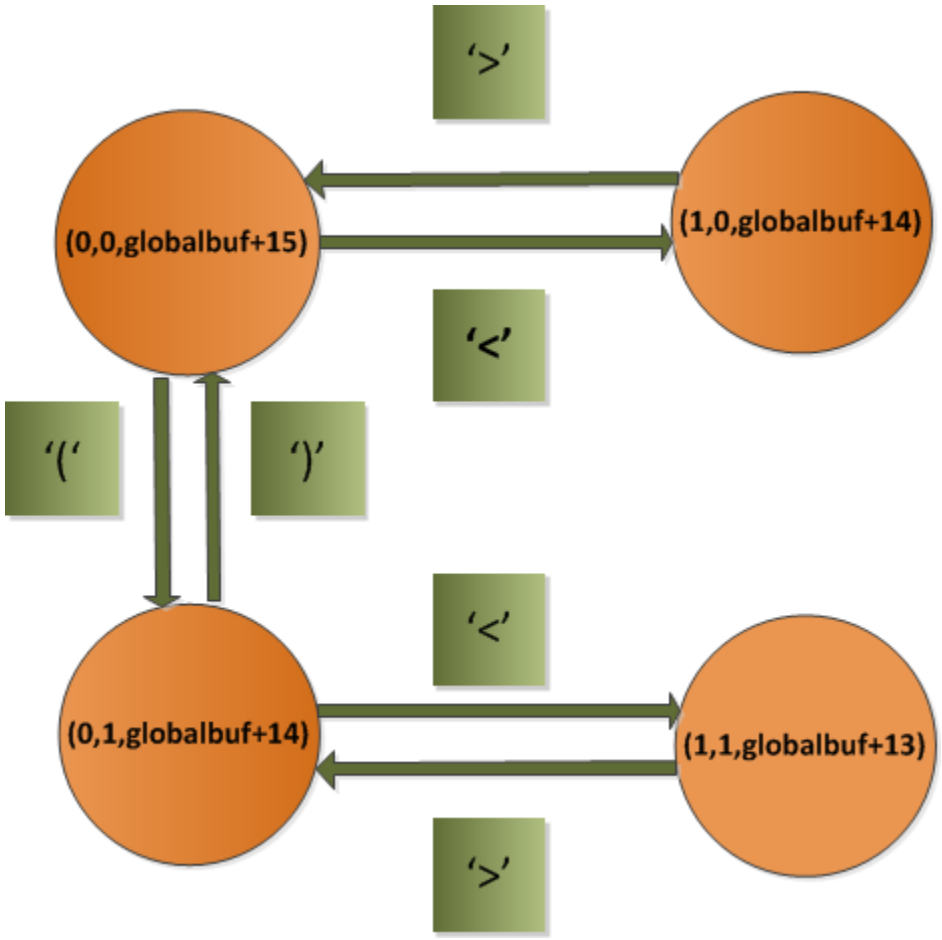
```
copy_it( char * input ){
  char localbuf[ BUFFERSIZE ];
  char c, *p = input, *d = &localbuf[0];
  char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
  int quotation = FALSE;
  int roundquote = FALSE;

  memset( localbuf, 0, BUFFERSIZE );
  while( (c = *p++) != '\0' ){
        if(( c == '<' ) && (!quotation)){
              quotation = TRUE;
              upperlimit--;}
        if(( c == '>' ) && (quotation)){
              quotation = FALSE;
              upperlimit++;}
        if(( c == '(' ) && ( !quotation ) && !roundquote){
              roundquote = TRUE;
              /*upperlimit--;*/}
        if(( c == ')' ) && ( !quotation ) && roundquote){
              roundquote = FALSE;
              upperlimit++;}
        // If there is sufficient space in the buffer, write the
        if( d < upperlimit )
              *d++ = c;
  }
  if( roundquote )
        *d++ = ')';
  if( quotation )
        *d++ = '>';

  printf("%d: %s\n", (int)strlen(localbuf), localbuf);
```



An inductive invariant for crackaddr()

Let us construct the finite state machine for this loop in domain (quotation,roundquote,offset) :

- **States correspond to memory values at the beginning of a loop iteration.**
- **Transitions correspond to executing an iteration after reading a character in the input string.**



We can feed this invariant to HAVOC in this syntax:

```
__loop_assert( (upperlimit == localbuf + 15 &&  quotation == FALSE && roundquote == FALSE)    ||
               (upperlimit == localbuf + 14 &&  quotation == TRUE && roundquote == FALSE)    ||
               (upperlimit == localbuf + 14 &&  quotation == FALSE && roundquote == TRUE)    ||
               (upperlimit == localbuf + 13 &&  quotation == TRUE && roundquote == TRUE) )
```

Checking example 3

```
c:\havoc-old\esp>c:\Boogie\Boogie.exe copy_it.bpl
Boogie program verifier v2.2.30705.1126,
Copyright (c) 2003-2011, Microsoft.
copy_it.bpl(522,1): This loop invariant might not be
maintained by the loop.
Execution trace:
   copy_it.bpl(418,1): start
   copy_it.bpl(518,1): label_16_head
   copy_it.bpl(548,1): label_17_true
   copy_it.bpl(563,1): label_18_true
   copy_it.bpl(583,1): label_19_false
   copy_it.bpl(595,1): label_21
   copy_it.bpl(604,1): label_21_false
   copy_it.bpl(637,1): label_25
   copy_it.bpl(646,1): label_25_false

[ Introduce fix ]

c:\havoc-old\esp>c:\Boogie\Boogie.exe copy_it.bpl
Boogie program verifier 2.2.30705.1126, Copyright
(c) 2003-2011, Microsoft.
Boogie program verifier finished with 1 verified, 0
errors
c:\havoc-old\esp>
```

- We run Boogie on the vulnerable code: loop invariant is found to be violated.
- We uncomment the fix line: the loop invariant is verified.
- Boogie/z3 can prove such loop invariants when they are **inductive** :
  - It is provable at the loop entry state
  - When provable at iteration N, then provable at iteration N + 1

➔ It is provable at any iteration

- On next slide, we show a more concise (abstract) invariant for the *crackaddr* loop. The simpler our invariants are, the most likely we can generate them automatically. Unfortunately, the more concise invariant is not provable by induction.

A non-inductive failing invariant

Let us try to find a more concise and elegant invariant for the loop that can capture the correct behavior. The blue line abstracts the two green lines in the new invariant. This introduces a new satisfying valuation of the formula with *upperlimit* offset 14 at the same time as one of *quotation* and *roundquote* variables are true (including when both are true at the same time).

- **Original (working) invariant:**

```
__loop_assert(
(upperlimit == localbuf + 15 &&  quotation == FALSE && roundquote == FALSE)     ||
(upperlimit == localbuf + 14 &&  quotation == TRUE && roundquote == FALSE)      ||
(upperlimit == localbuf + 14 &&  quotation == FALSE && roundquote == TRUE)      ||
(upperlimit == localbuf + 13 &&  quotation == TRUE && roundquote == TRUE) )
```

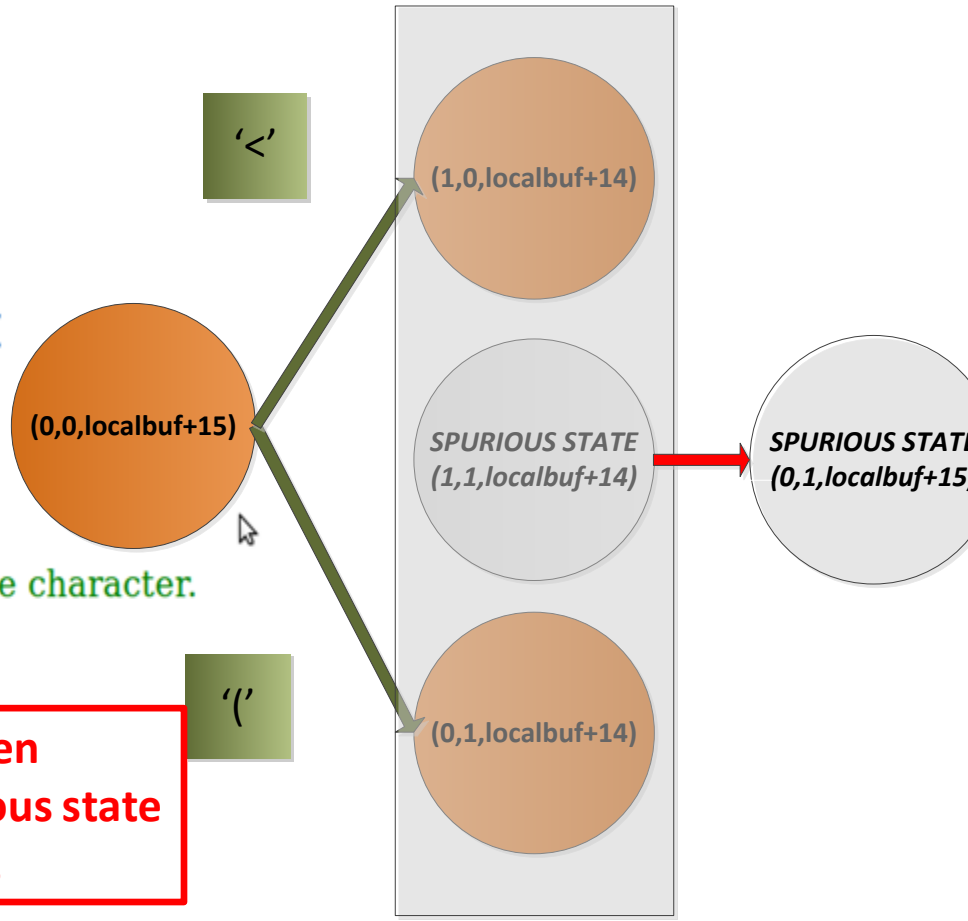
- **More concise (abstract) invariant : does loop verification still work?**

```
__loop_assert(
(upperlimit == localbuf + 15 &&  quotation == FALSE && roundquote == FALSE)   ||
(upperlimit == localbuf + 14 &&  (quotation == TRUE || roundquote == TRUE))   ||
(upperlimit == localbuf + 13 &&  quotation == TRUE && roundquote == TRUE)  )
```

ample_bad.c

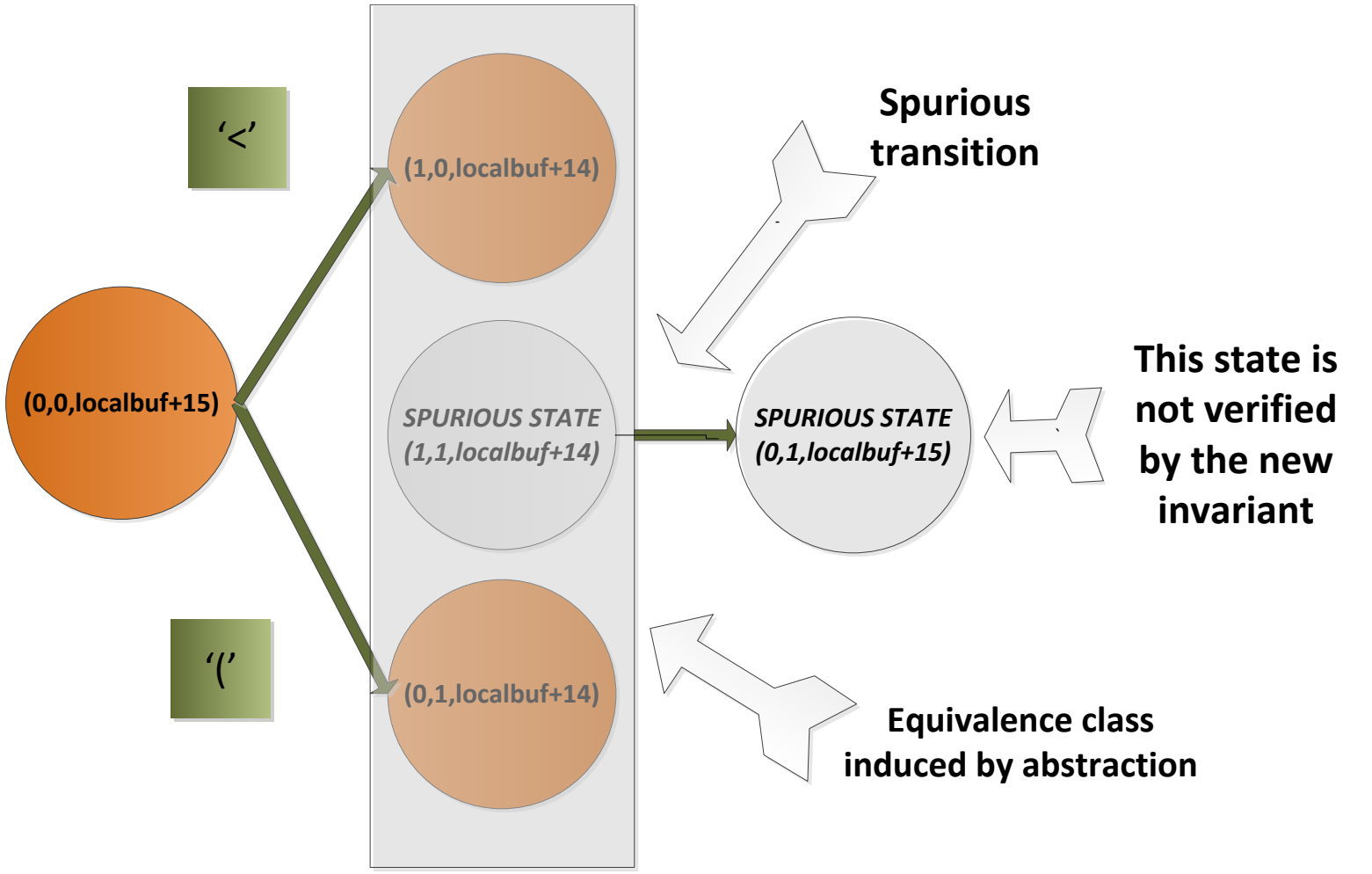
```
copy_it( char * input ){
  char localbuf[ BUFFERSIZE ];
  char c, *p = input, *d = &localbuf[0];
  char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
  int quotation = FALSE;
  int roundquote = FALSE;

  memset( localbuf, 0, BUFFERSIZE );
  while( (c = *p++) != '\0' ){
        if(( c == '<' ) && (!quotation)){
              quotation = TRUE;
              upperlimit--;}
        if(( c == '>' ) && (quotation)){
              quotation = FALSE;
              upperlimit++;}
        if(( c == '(' ) && ( !quotation ) && !roundquote){
              roundquote = TRUE;
              /*upperlimit--;*/}
        if(( c == ')' ) && ( !quotation ) && roundquote){
              roundquote = FALSE;
              upperlimit++;}
        // If there is sufficient space in the buffer, write the character.
        if( d < upperlimit )
              *d++ = c;
  }
  if( roundquote )
        *d++ = ')';
  if( quotation )
        *d++ = '>';

  printf("%d: %s\n", (int)strlen(localbuf), localbuf);
```

__loop_assert(
(upperlimit == localbuf + 15 && quotation == FALSE && roundquote == FALSE)
(upperlimit == localbuf + 14 && (quotation == TRUE || roundquote == TRUE))
(upperlimit == localbuf + 13 && quotation == TRUE && roundquote == TRUE))



The red transition would happen if the loop started in the spurious state and executed the red iteration.

A non-inductive failing invariant



This invariant is NOT inductive/verified due to the spurious transition (T,T,+14) → (F,T,+15)

```
__loop_assert( (upperlimit == localbuf + 15 &&  quotation == FALSE && roundquote == FALSE)  ||
               (upperlimit == localbuf + 14 &&  (quotation == TRUE || roundquote == TRUE))  ||
               (upperlimit == localbuf + 13 &&  quotation == TRUE && roundquote == TRUE)
```

Taking back Halvar's list

- Most abstract interpretation-style analyses will try to map program lines to sets of states for variables. When control flow converges, states are merged and "safely approximated" such as *"state 00 and p between 0 and 4"* combined with *"state 01 and p between 0 and 2"* will be combined into *"state 00 or 01 and p between 0 and 4"* . This contains spurious states: 01 and p=4 can't actually happen. More precision is lost on each iteration of the loop.

- When we could solve all the inter-procedural analysis and C++ issues, we still fail on heavily simplified versions of real-world code

Taking back Halvar's list

~~• Most abstract interpretation-style analyses will try to map program lines to sets of states for variables. When control flow converges, states are merged and "safely approximated" such as *"state 00 and p between 0 and 4"* combined with *"state 01 and p between 0 and 2"* will be combined into *"state 00 or 01 and p between 0 and 4"* . This contains spurious states: 01 and p=4 can't actually happen. More precision is lost on each iteration of the loop.~~

~~• When we could solve all the inter-procedural analysis and C++ issues, we still fail on heavily simplified versions of real-world code~~

None of these limitations hold
if you sacrifice 100% automation

Think Cyborg, not Robot

Take a step back…

- Lesson learned:
  - Invariants capture the correct (expected) behaviors of programs.
  - A vulnerability is a deviation of this invariant. The deviation is unknown and unspecified by the invariant.
- Quid of the "*weird machine*" ?
  - Sergey Bratus coined the term *weird machine* to refer to the staged (stateful) vulnerability exploitation mechanisms leading to untrusted code execution.
  - The presented analysis methodology aims at finding what are the initial states of the weird machine. It does not try to take any transition of it. Exploitation is a different task than bug finding.

Conclusion

- Static analysis is a practical technique used today to find a large number of vulnerabilities
  - One application: variation analysis
- There is a trade-off between full automation and expressiveness & precision of analysis
  - Analyst injects domain specific knowledge
- Beyond current tools, there is room for impact improvement and more practical research
  - Get your hands dirty

Acknowledgments

- Thomas Ball and Shaz Qadeer at Microsoft Research for all their mentoring and support on the HAVOC project.
- Mark Dowd and Chris Rohlf for finding the vulnerabilities used in this presentation.
- Sean Heelan and Thomas Dullien for popularizing the vulnerabilities and inviting us to work on them.
- The great people of Microsoft Security (MSEC / MSRC)

Microsoft is hiring

Join us to build computer security for the next ten years at:

https://careers.microsoft.com/search.aspx#&&p4=all&p0=MSEC+MSRC&p5=all&p1=all&p2=all&p3=all

References

[1] Revisiting Precise Program Verification using SMT Solvers
(S.K.Lahiri, S.Qadeer), POPL'08
[2] ExplainHoudini: Making Houdini inference transparent
(S.K.Lahiri, J.Vanegue), VMCAI'11
[3] Zero allocations vulnerabilities
(J.Vanegue), Usenix Security WOOT'10
[4] The Future of exploitation revisited
(T.Dullien), Infiltrate security conference 2011
[5] Vulnerability Detection Systems: Think Cyborg, Not Robot
(S.Heelan), IEEE S&P journal volume 9 issue 3
[6] Remote Sendmail Header Processing Vulnerability
(M.Dowd), ISS X-force advisory 142, March 2003
[7] WebKit CSS Font Face Parsing Type Confusion
(C.Rohlf), em386.blogspot.com, November 2010
[8] SMT solvers for software security
(J.Vanegue,S.Heelan,R.Rolles), Usenix Security WOOT'12 (to appear)

We want to hear your questions and feedback!

jvanegue@microsoft.com
shuvendu@microsoft.com

BONUS SLIDES

Under the hood: Boogie IR

- **Based on the dynamic single assignment (DSA) intermediate form**
  - Every assignment creates a fresh version number for the destination variable (as in Static Single Assignment form i.e. SSA)
    Ex: a1 = 42; if (a1) { a2 = a1 + 1; } else { a3 = a1 + 2; }
  - In DSA, more than one assignment can be done on the same versioned variable as long as this cannot happen at run time (Previous example can use a2 in both branches in DSA, not in SSA).

- **Use assume/assert logic explicitly encoded in analyzed program.**
  - **Assume** introduces a new assumption (ex: a path condition)
    - Automatically inserted at the beginning of basic blocks by HAVOC to reflect the augmented guarding condition.
    - Assume(x) = true means the set of assumptions is consistent.
    - Assume(x) = false means the path is infeasible. This happens when the assumptions are contradictory.
  - **Assert** checks if a certain condition is true at program location
    - Guided by user, inserted where properties have to be checked.
    - Assert(x) = true means everything is alright (X is true, analysis continues, no violation can happen on this path)
    - Assert(x) = false leads to a static analysis alert.

HAVOC C++ to Boogie translation

- Use new function (now method) name scheme embedding the class name. i.e. MethodName$ClassName instead of FunctionName.
- Adapted instrumentation scheme to work with methods and objects: __instrument_call_pre(MethodName$*, *$ClassName, …)
- Dynamic dispatch boogie IR generation to support virtual table calls on derived methods in object hierarchy. All 20+ operators are treated as regular methods that can be overloaded.
- Support C++ references as syntactic sugar for accesses on pointed data structures. Allow the presence of C++ references in instrumentations.
- Boogie translation is performed after the C++ compiler template specialization. The prover only sees specialized classes in the program IR.
- Unsupported (as of Nov 2011): anonymous functions (lambdas), automated dynamic type inference, parametric polymorphism (WIP: make use of unique identifiers for every instance of named methods still sharing the same name after all previous compiler transformations)

Open challenges in practical analysis

1. Type safety for C/C++
   - In C : Propagate type information in the presence of unsafe casts
   - In C++ : Propagate dynamic type information (application: precise static vtable lookups)
   - *Flow (in)sensitive type qualifiers* by J Foster, R Johnson, J Kodumal, A Aiken http://www.cs.umd.edu/~jfoster/papers/toplas-quals.html

2. Invariant synthesis
   - *Precondition Inference from Intermittent Assertions and Application to Contracts on Collections* by P Cousot, R Cousot, F Logozzo (VMCAI 2011) http://www.di.ens.fr/~cousot/COUSOTpapers/VMCAI-11.shtml

3. Concurrent programs analysis
   - Interleaving leads to state space explosion
   - Partial Order Reduction: http://en.wikipedia.org/wiki/Partial_order_reduction
   - The Poirot tool: http://research.microsoft.com/pubs/148752/tr.pdf (MSR)

4. Test generation (Input crafting)
   - Automatically create a witness test to confirm the property violation.
   - *Automatic generation of control flow hijacking exploits* by S Heelan
   - *http://seanhn.files.wordpress.com/2009/09/thesis1.pdf*