SMT solvers for software security

Tales of automation…

Usenix Security Workshop on Offensive Technologies (WOOT'12)

August 7th 2012, Bellevue, WA, USA

Julien Vanegue (Microsoft Security Science)

Sean Heelan (Immunity Inc.)

Rolf Rolles (Unaffiliated)

Summary

- Discuss the applicability of SMT (Satisfiability Modulo Theories) solvers for software security, in particular for:
  - Vulnerability checking
  - Exploit generation
  - Copy protection analysis
- Overall workflow: abstract a program into a formula and check if the formula has the desired properties
- Overall message: SMT solvers perform great once the problem domain has been defined. Solvers do not define the problem automatically.

Part I: Finding vulnerabilities using SMT

Work flow for static security checking

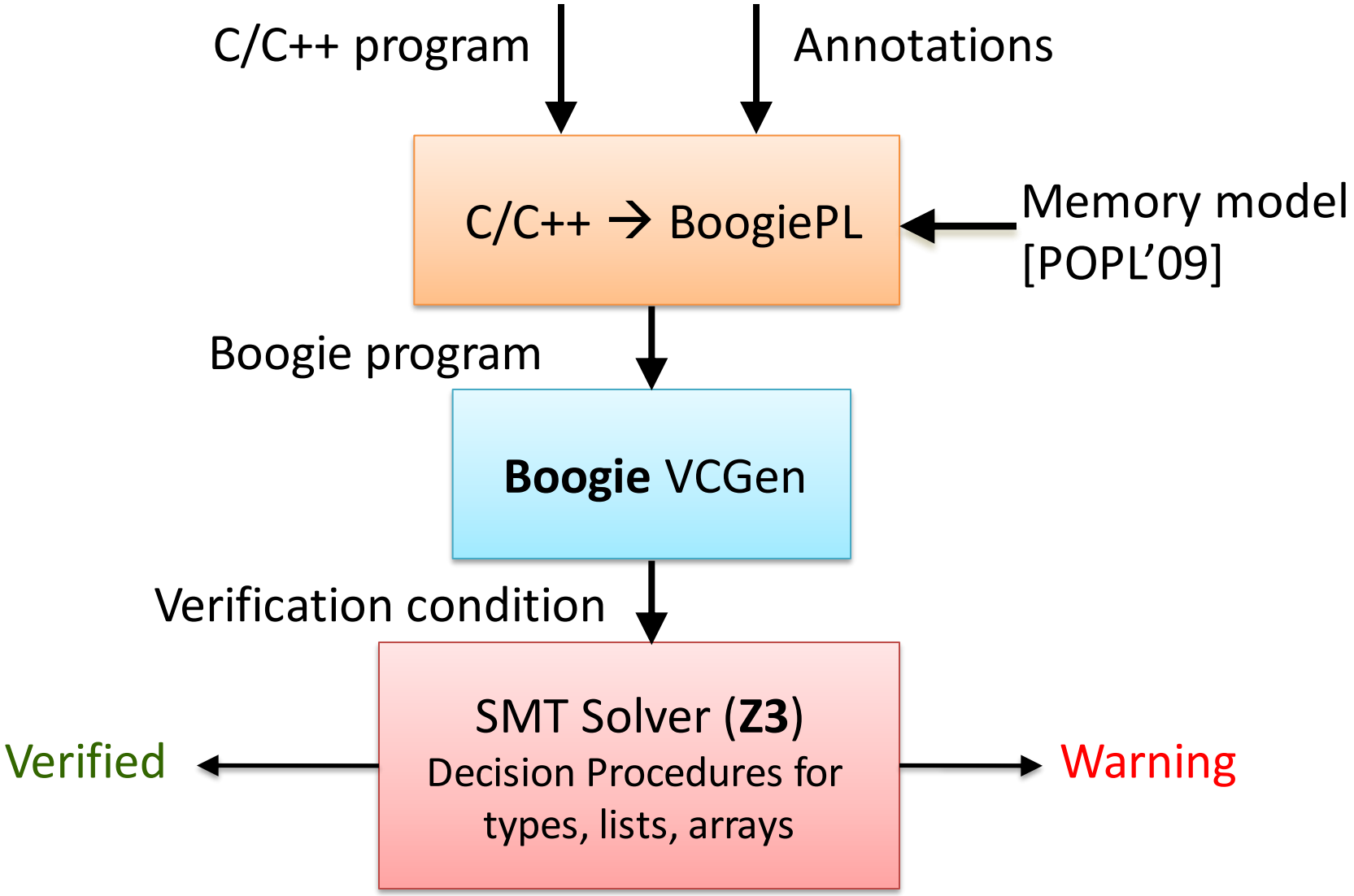
(1) The security auditor formalizes what he thinks is the formal code contract into an annotation (directive to the checker)

(2) The analyzed source code gets translated into an intermediate form.

(3) The intermediate form is consumed by a theorem prover that creates a “verification condition” (VC, aka a safety formula)

(4) The VC is passed to the SMT solver for resolution

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



Example on Loop analysis
Sendmail CrackAddr() buffer overflow

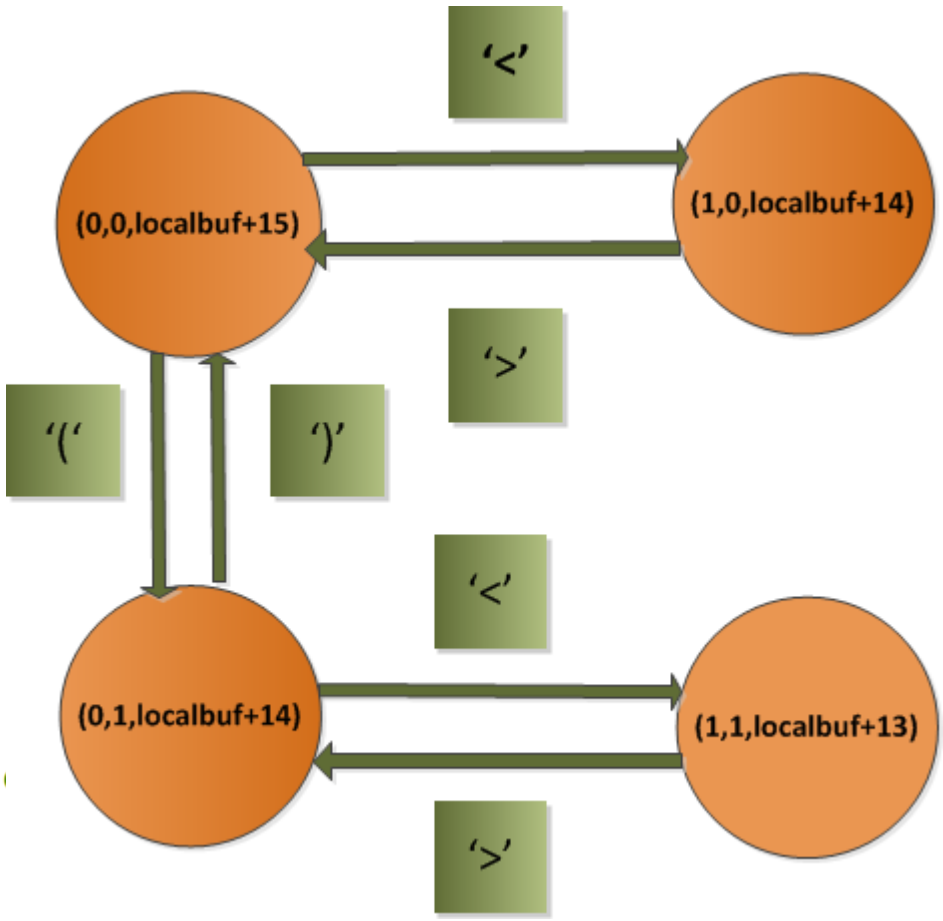
- CVE-2002-1337: "*A buffer overflow in sendmail 5.79 to 8.12.7 allows remote attackers to execute arbitrary code via certain formatted address fields, related to sender and recipient header comments as processed by the crackaddr function of headers.c*"
- Published by Mark Dowd, Exploited by Last Stage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

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);
```

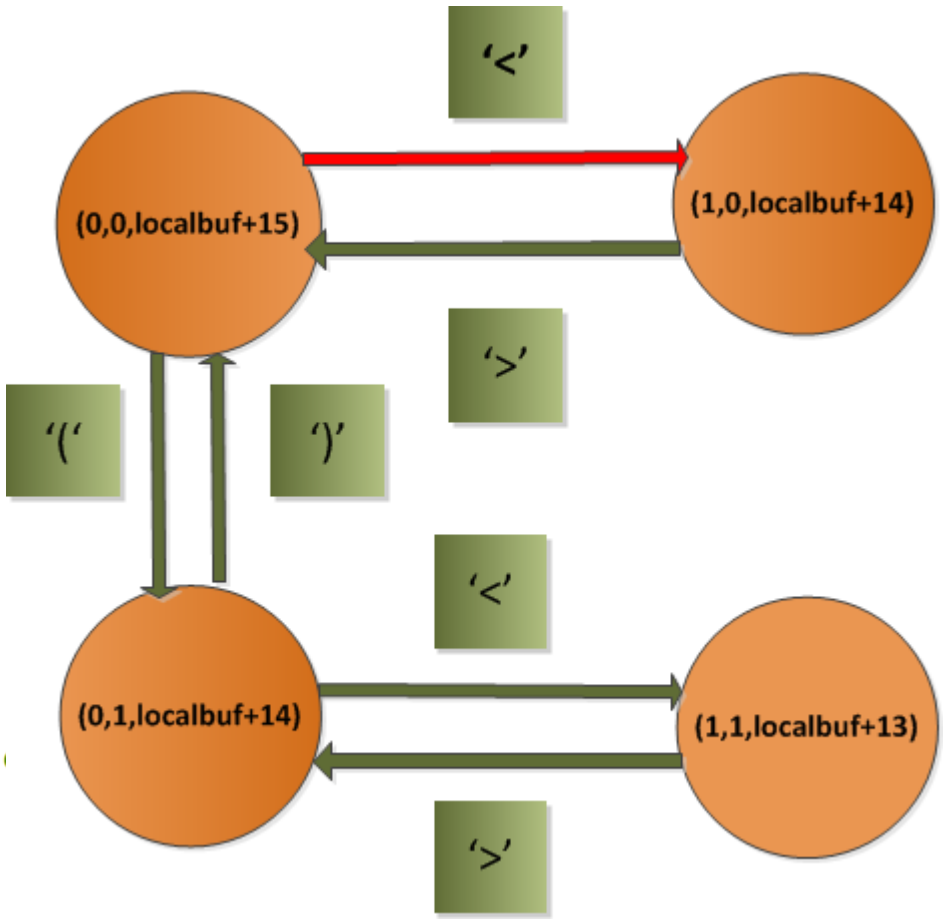
ample_bad.c

```
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);
```



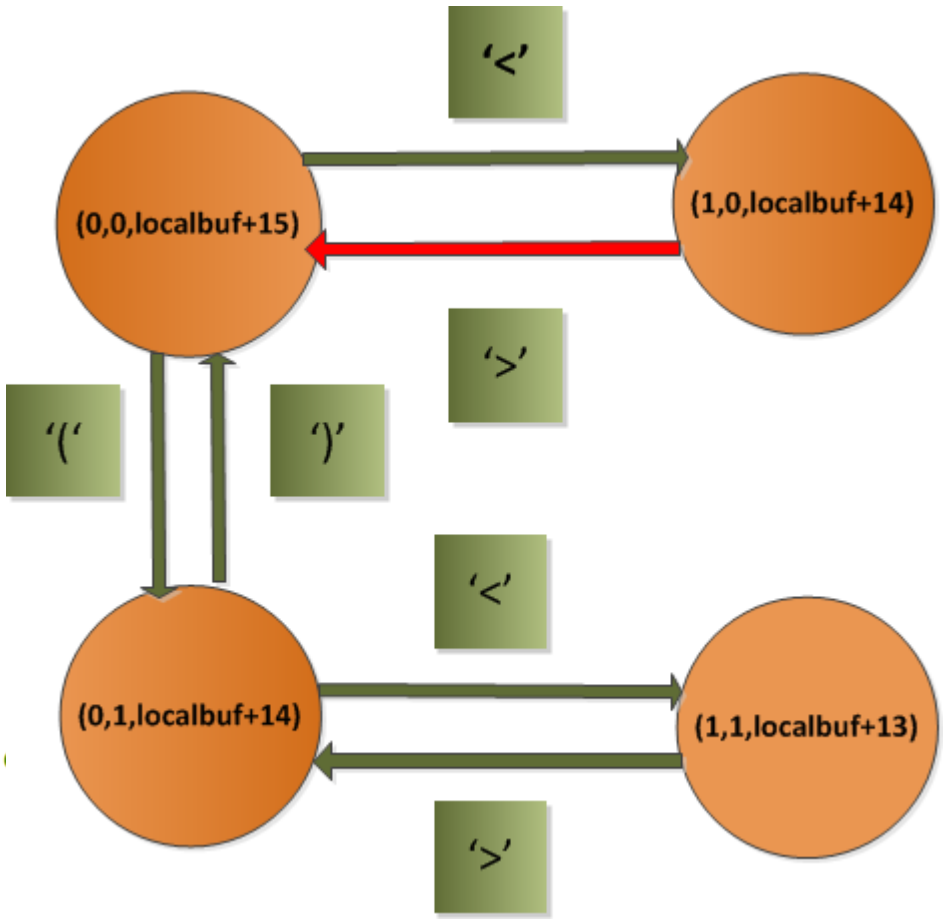
ample_bad.c

```
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);
```

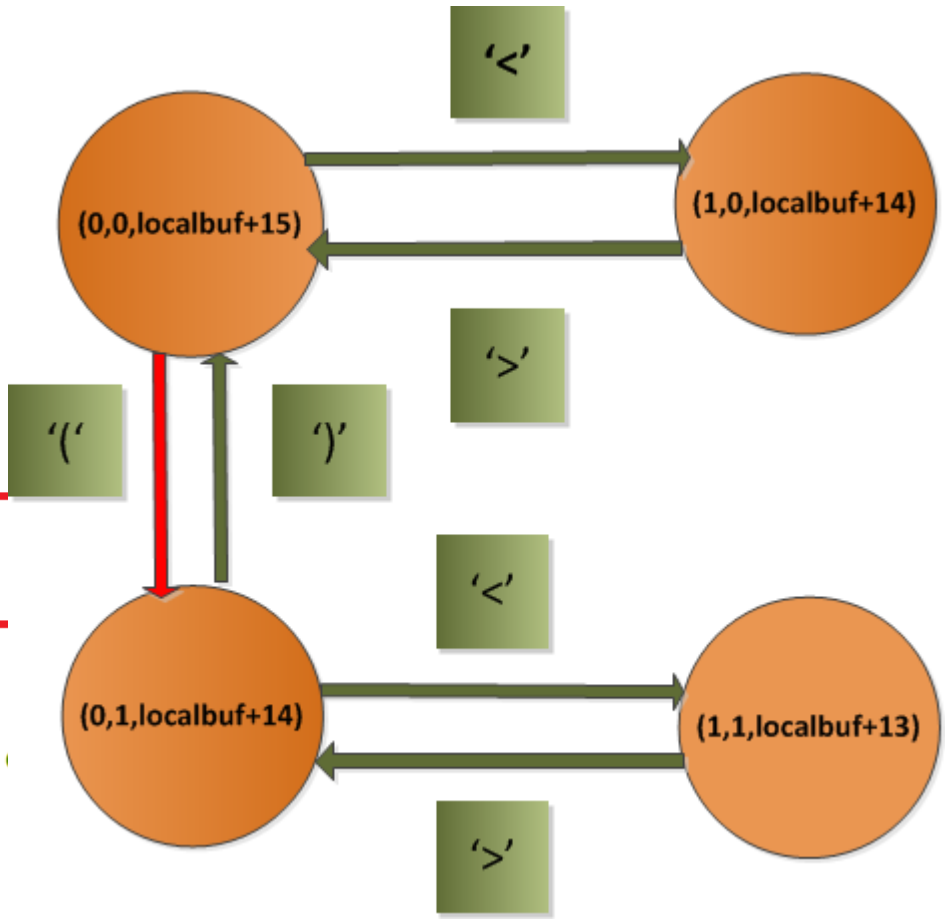


ample_bad.c

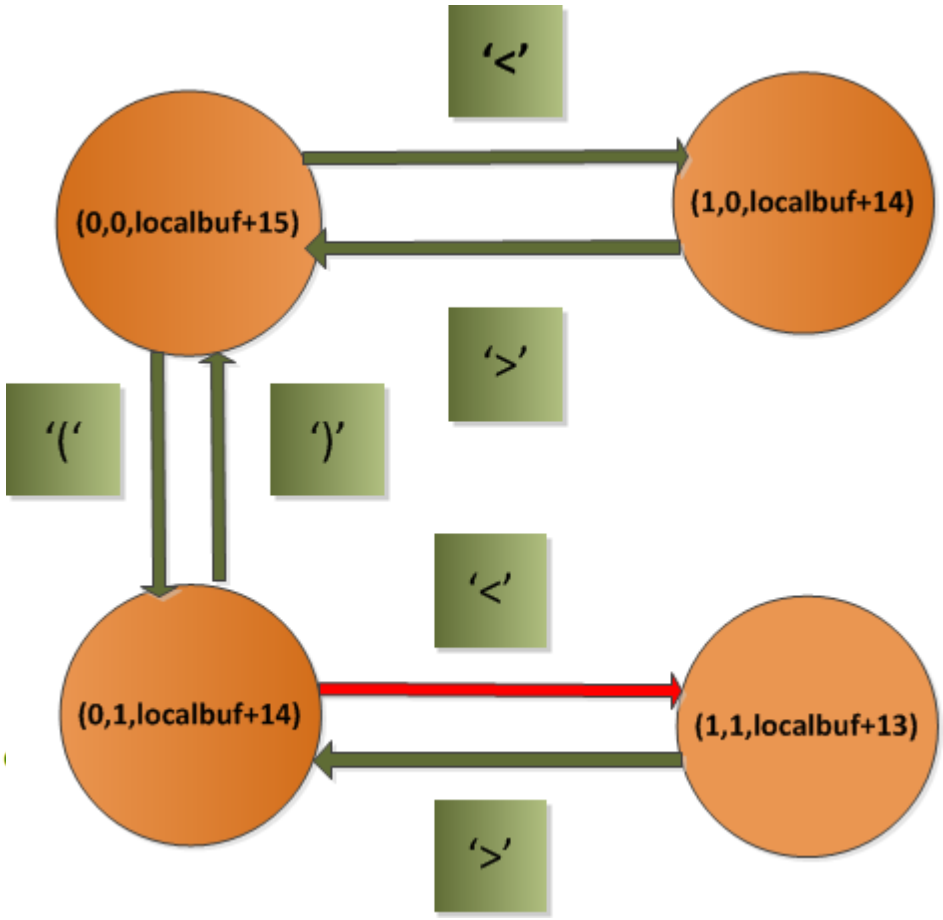
```
copy_it( char * input ){
  char localbuf[ BUFFERSIZE ];
  char c, *p = input, *d = &localbuf[0];
  char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
  int quotation = FALSE;
  int roundquote = FALSE;

  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);
```



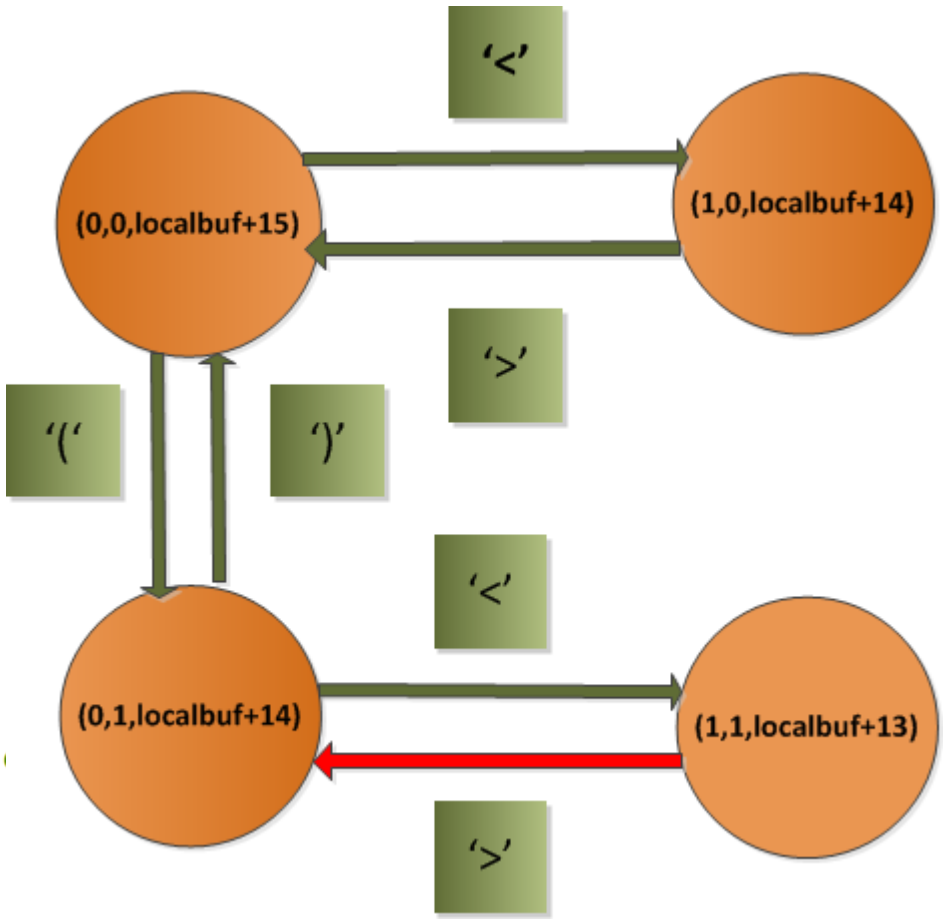
```
copy_it( char * input ){
  char localbuf[ BUFFERSIZE ];
  char c, *p = input, *d = &localbuf[0];
  char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
  int quotation = FALSE;
  int roundquote = FALSE;

  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);
```



```
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);
```

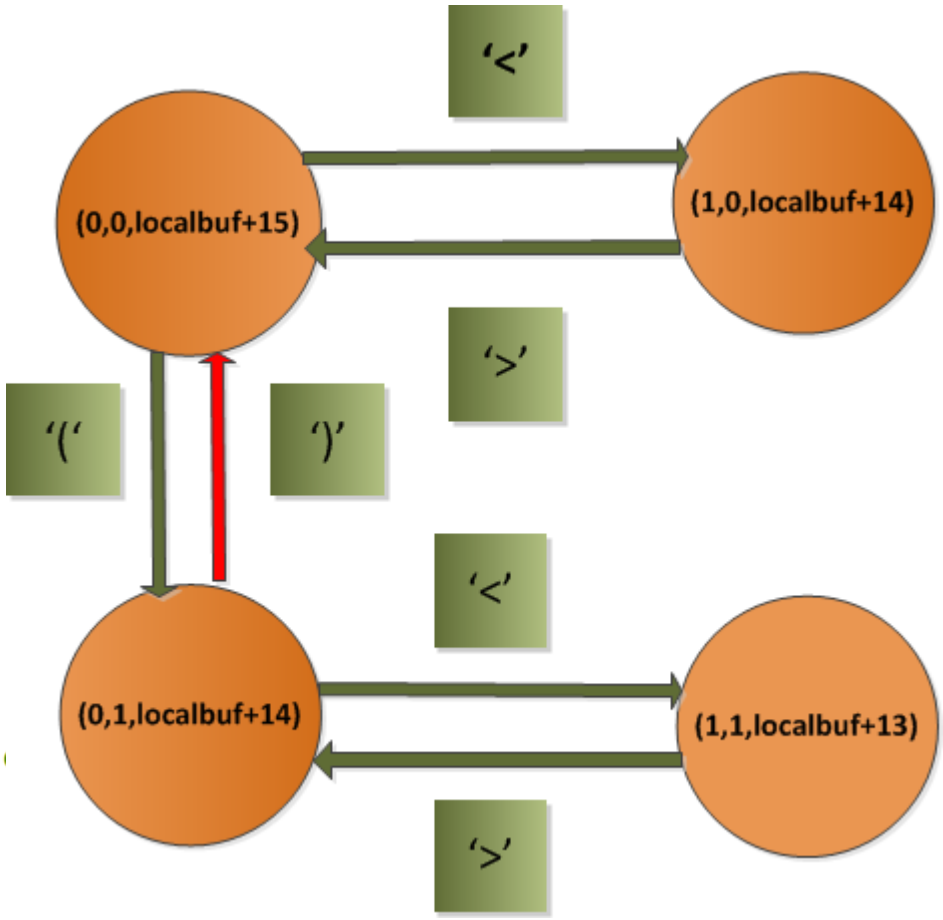


```
copy_it( char * input ){
  char localbuf[ BUFFERSIZE ];
  char c, *p = input, *d = &localbuf[0];
  char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
  int quotation = FALSE;
  int roundquote = FALSE;

  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);
```



ample_bad.c

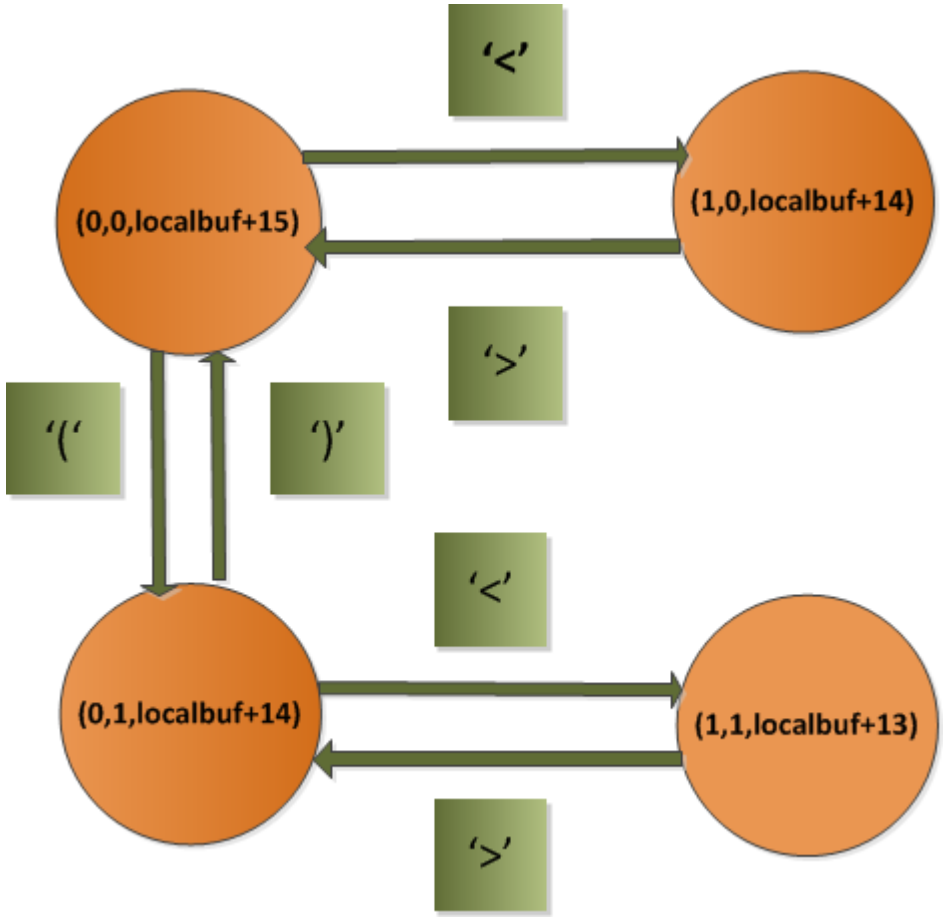
```
copy_it( char * input ){
  char localbuf[ BUFFERSIZE ];
  char c, *p = input, *d = &localbuf[0];
  char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
  int quotation = FALSE;
  int roundquote = FALSE;

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

Part II: Assisting exploit generation using SMT

Automatic Exploit Generation (AEG)

- Loosely defined as *"Given a program and a vulnerability, automatically craft an input that redirects control flow to malicious code"*
- Canonical example of a problem domain where there have been many successes but real world applicability is hindered by modeling/constraint generation rather than solver performance or features

State of the Art

- Based directly on techniques and platforms built for symbolic/concolic execution
- Given a sequence of instructions, construct a **path condition** reflecting the **modifications** (e.g. arithmetic transformations) and **constraints** (e.g. conditional jumps) on attacker provided input

Example: Constructing the Path Condition

```
0: add al, al
1: sub al, 0x0f
2: test al, al
3: jz 6
4: …
5: jmp 7
6: …
```

Path Condition at 6:

$$a_1 = a_0 + a_0$$

$$\wedge\ a_2 = a_1 - 15$$

$$\wedge\ a_2 = 0$$

(a_0 represents the initial input with a_n created on a register write)

Restricted-Model Exploit Generation

- The entirety of knowledge possessed by the AEG system is in the form of these path conditions
- Lets represent this as a map **K** from registers/memory locations to logical formulae: **K: M → F**

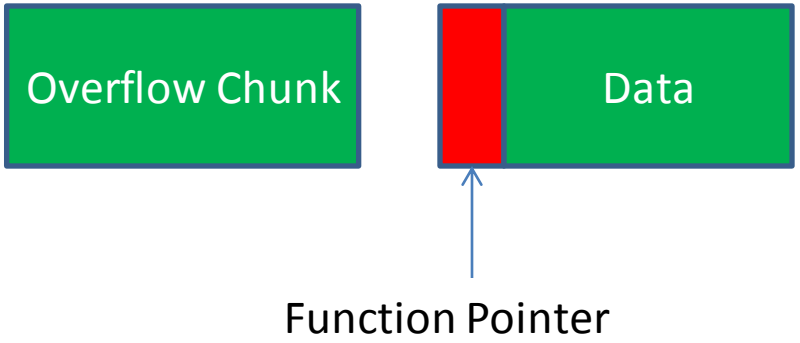
  Ex: *M[eax] = (eax != 0 ^ Tainted(eax))*

- No information on the relationship between user input and the heap state, thread execution, signals/events etc

Restricted-Model Exploit Generation

Heap Layout on Run / Input 1

Heap Layout on Run / Input 2



(1) What can **K** tell us about the data contained in the overflow chunk and other heap chunks?

(2) What can **K** tell us about the effect of attacker input on the relative layout of the overflow chunk with respect to other heap chunks?

Accurate exploit generation requires information from both categories but K only provides the first

Restricted-Model Exploit Generation

- **K** combined with a set of *exploit templates* to produce SMT formulae
  - A template is a static set of rules describing the constraints to generate when certain conditions are met (ex: "*If the stored return address can be influenced then try to set it to address 0xABCD")*
  - Crucially, a template is restricted to the information present in **K** (no heap crafting or other external environment control strategies)
- At present templates have only described basic, single-shot exploitation techniques (no vulnerability chaining, no information disclosures)

Deconstructing the myth of fully automated exploit generation

- The real execution environment is **not** accurately described by **K**
- Modern operating systems have killed the majority of generic exploitation techniques
  - More often than not static templates are not a feasible solution (**need domain specific language?)**
- Usually, modern exploits require application specific techniques, memory crafting and bug chaining: **how to encode this in SMT?**

Part III: Analyze copy protection using SMT

Copy protection analysis

- SMT solvers for copy protection analysis have been used mostly in two scenarios:

  (1) Equivalence checking between obfuscated code generated out of a copy protection technology and the original non-obfuscated code.

  (2) Semi-automated cryptanalysis of serial number algorithms and bypassing them using key generation (*"keygens"*)

- We give an example of (2) in next slide

Ex: the main loop for a serial algorithm

1. again:
2. lodsb // $al_{0,i}$ = activation code[i]
3. sub al, bl // $\wedge\ al_{1,i} = al_{0,i} - ebx_i[7:0]$
4. xor al, dl // $\wedge\ al_{2,i} = al_{1,i} \oplus edx_i[7:0]$
5. Stosb // $\wedge$ output[i] = $al_{2,i}$
6. rol edx, 1 // $\wedge\ edx_{i+1}$ = rotate_left(edx_i, 1)
7. rol ebx, 1 // $\wedge\ ebx_{i+1}$ = rotate_left(ebx_i, 1)
8. loop again

Conclusion

- SMT solvers have demonstrated they can effectively resolve constraints generated by static security checkers, AEG systems, and copy protection analysts.
- Improving the state of the art will require more effective modeling of the execution environment and more sophisticated methods for generating candidate formulae without user-interaction (automated constraint generation a.k.a. inference)
- Interesting challenges remain in encoding more complex and informed strategies as SMT formula (ex: to model exploit mitigation bypass techniques)