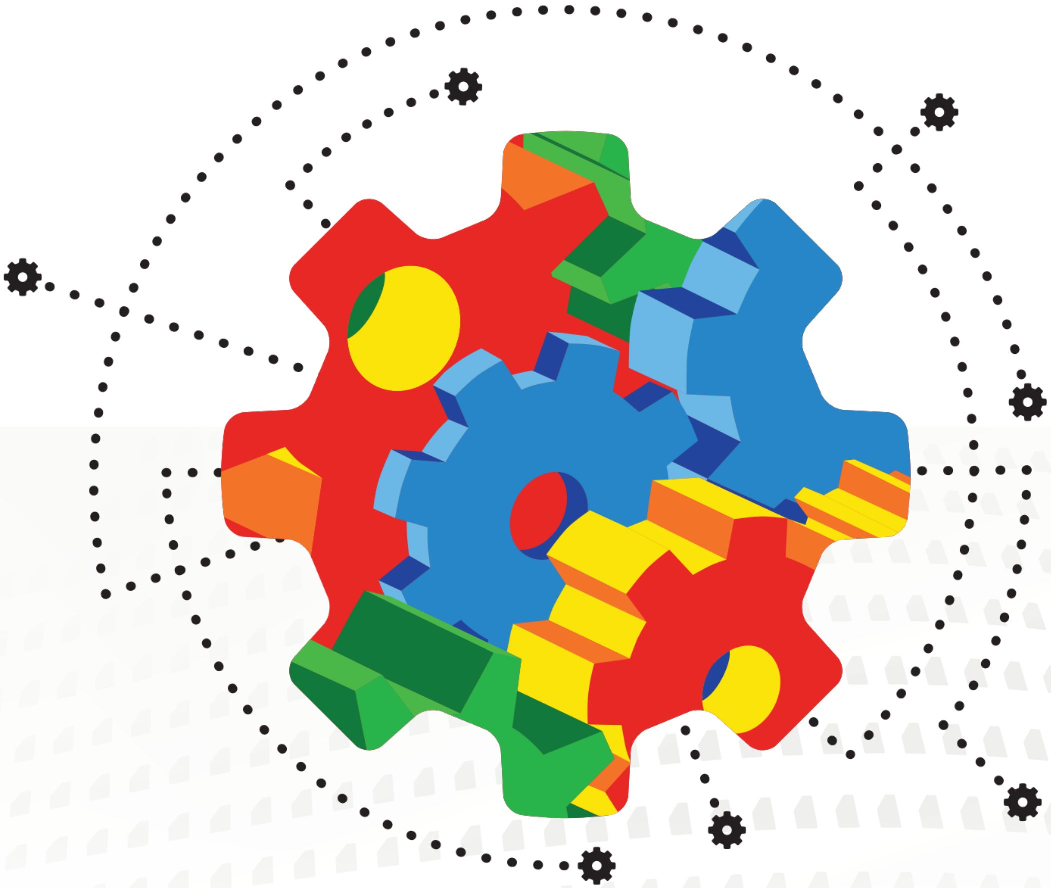
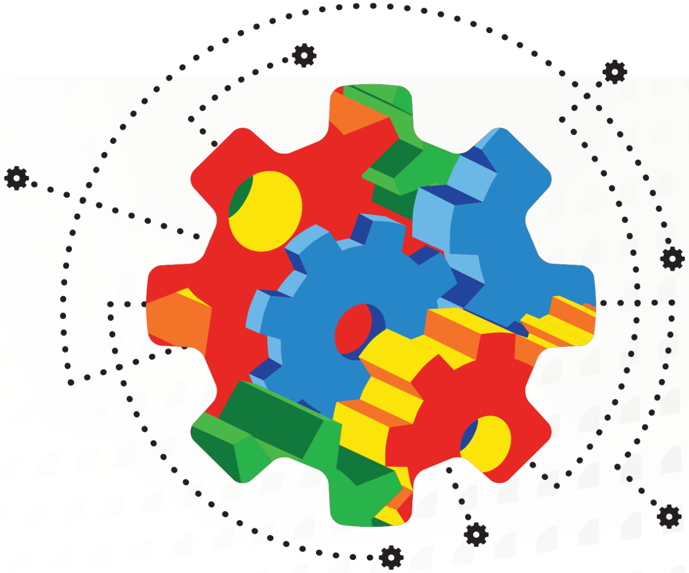


Fuzzing and Patch Analysis: SAGEly Advice

Sourcefire is now part of Cisco.



Introduction

Sourcefire is now part of Cisco.

Automated Test Generation

- Goal: Exercise target program to achieve full coverage of all possible states influenced by external input

- Code graph reachability exercise
- Input interaction with conditional logic in program code determines what states you can reach

Automated Testing Approaches

- Modern approaches fall into two buckets:
  - → Random Testing (Fuzzing)
    - Zero-knowledge mutation
    - Syntax model based grammar
    - Direct API interrogation
  - → Concolic Testing
    - Instrumented target program
    - Tracking of dataflow throughout execution
    - Observation of program branch logic & constraints
    - Symbolic reasoning about relationship between input and code logic

Advanced Mutation Fuzzing

- Advanced mutation fuzzers derive grammars from well formed data samples or are given a manually constructed syntax & interaction model that is expressed in a higher level grammar
- For automation, syntax is inferred using string grouping algorithms such as n-gram
- A good modern example is Radamsa
  - → Supply a corpus of well formed inputs
  - → Multiple grammar inference strategies
  - → Detection of repeated structures or identification of basic types is automatic

Limits to Fuzzing

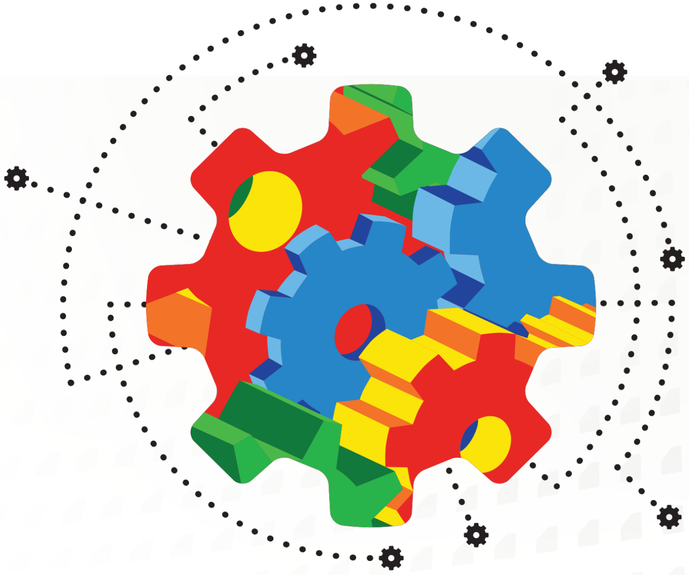
- Unfortunately even the most advanced fuzzers cannot cover all possible states because they are unaware of data constraints.
- The below example would require an upper bound of 2^32 or 4 billion attempts to meet the condition required to trigger the crash

```
void test(char *buf)
{
    int n=0;
    if(buf[0] == 'b') n++;
    if(buf[1] == 'a') n++;
    if(buf[2] == 'd') n++;
    if(buf[3] == '!') n++;
    if(n==4) {
        crash();
    }
}
```

Concolic Testing

- For anything beyond string grouping algorithms, direct instrumentation of the code and observation of interaction between data and conditional logic is required

- Early academic work in this area:
  - → DART: Directed Automated Random Testing
    - 2005 - Patrice Godefroid, et al
  - → CUTE: a concolic unit testing engine for C
    - 2005 – Koushik Sen
  - → EXE: Automatically Generating Inputs of Death
    - 2006 - Dawson Engler



Concolic Test Generation: Core Concepts

Sourcefire is now part of Cisco.

Code Coverage & Taint Analysis

- Code Coverage
  - → Analysis of program runtime to determine execution flow
  - → Collect the sequence of execution of basic blocks and branch edges

- Several approaches
  - → Native debugger API
  - → CPU Branch Interrupts
  - → Static binary rewriting
  - → Dynamic binary instrumentation

Code Coverage & Taint Analysis

- Taint Analysis
  - → Analysis of program runtime to determine data flow from external input throughout memory
  - → Monitor each instruction for propagation of user controlled input from source operands to destination operands
  - → Dependency tree is generated according to tainted data flows in memory or CPU registers
  - → Taint analysis is imperfect – propagation rules must dictate the level of inferred dataflow that is propagated

Dynamic Binary Instrumentation

- JIT modification of binary code
  - → As new code blocks are visited or modules are loaded, an analysis phase disassembles the binary to identify code structure
  - → Instructions may be inserted at arbitrary locations around or within the disassembled target binary
  - → Modified code is cached and referenced instead of original binary

- Skips some problems with static binary rewriting and maintains runtime state for conditional instrumentation

Symbolic Execution

- Symbolic execution involves computation of a mathematical expression that represents the logic within a program
- It can be thought of as an algebra designed to express computation

```
void test(char *buf)
{
    int n = 0;
    if(buf[0] == 'b') n++;
    if(buf[1] == 'a') n++;
    if(buf[2] == 'd') n++;
    if(buf[3] == '!') n++;
    if(n==4) {
        crash();
    }
}
```

```
(declare-const buf (Array Int Int))
(declare-fun test () Int)
(declare-const n Int)
(assert (= n 0))
(ite (= (select buf 0) 98) (+ n 1) 0)
(ite (= (select buf 1) 97) (+ n 1) 0)
(ite (= (select buf 2) 100) (+ n 1) 0)
(ite (= (select buf 3) 92) (+ n 1) 0)
(assert (= n 4))
(check-sat)
(get-model)
```

Symbolic Execution

- Symbolic execution involves computation of a mathematical expression that represents the logic within a program
- It can be thought of as an algebra designed to express computation

```
void condition(int x)
{
     int ret = 0;
     if (x >= 50)
          ret = 1;
     else
          ret = 2;
     return ret
}
```

```
(declare-fun condition () Int)
(declare-const ret Int)
(declare-const x Int)
(assert (=> (>= x 50) (= ret 1)))
(assert (=> (< x 50) (= ret 2)))
(assert (= ret 1))
(check-sat)
(get-model)
---
sat
(model
     (define-fun x () Int 50)
     (define-fun ret () Int 1)
)
```

Constraint Generation

- Comparisons are done on values to determine which branch of code to take:

```
if (a > b):
     block1
else:
     block2
```

- We observe these constraints to determine what data value ranges allow execution in different paths
- A code path is determined by collecting a series of these constraints which determines the execution flow of the program

Constraint Generation

- Against binary targets we need to track flags and evaluate the dependent comparison before a jump

```
0x080483d4 <+0>:     push   %ebp
0x080483d5 <+1>:     mov    %esp,%ebp
0x080483d7 <+3>:     and    $0xfffffff0,%esp
0x080483da <+6>:     sub    $0x10,%esp
0x080483dd <+9>:     cmpl   $0x1,0x8(%ebp)
0x080483e1 <+13>:    jle    0x80483f1 <main+29>
0x080483e3 <+15>:    movl   $0x80484d0,(%esp)
0x080483ea <+22>:    call   0x80482f0 <puts@plt>
0x080483ef <+27>:    jmp    0x80483f2 <main+30>
0x080483f1 <+29>:    nop
0x080483f2 <+30>:    leave
0x080483f3 <+31>:    ret
```

- This may be done manually or through the use of an IR

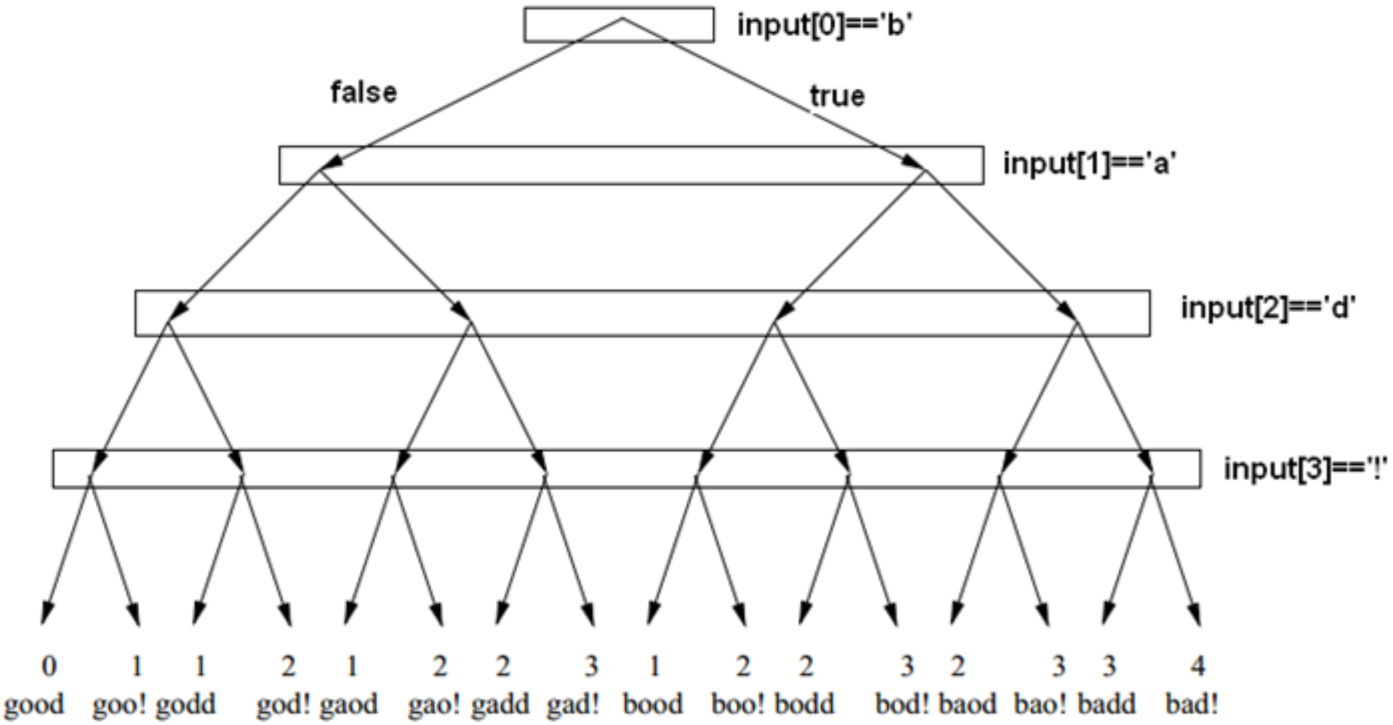
Constraint Solving

- A formula representing the code path logic is generated in a format acceptable to a symbolic execution engine

- To explore alternate paths, we invert the conditional logic of the last branch and allow the solver to generate an example that would match the inverted conditional logic

- Iterative use of this algorithm allows us to explore a complete program graph

Test Generation



Test Generation

- Input: ‘badd’
- Formula generated by symbolic execution:
  - → $\Phi := (i_0 = \text{'b'})\ \&\&\ (i_1 = \text{'a'})\ \&\&\ (i_2 = \text{'d'})\ \&\&\ (i_3 <> \text{'!'})$
- New formulas:
  - → $\Phi_0 := (i_0 = \text{'b'})\ \&\&\ (i_1 = \text{'a'})\ \&\&\ (i_2 = \text{'d'})\ \&\&\ (i_3 = \text{'!'})$
  - → $\Phi_1 := (i_0 = \text{'b'})\ \&\&\ (i_1 = \text{'a'})\ \&\&\ (i_2 <> \text{'d'})\ \&\&\ (i_3 <> \text{'!'})$
  - → $\Phi_2 := (i_0 = \text{'b'})\ \&\&\ (i_1 <> \text{'a'})\ \&\&\ (i_2 = \text{'d'})\ \&\&\ (i_3 <> \text{'!'})$
  - → $\Phi_3 := (i_0 <> \text{'b'})\ \&\&\ (i_1 = \text{'a'})\ \&\&\ (i_2 = \text{'d'})\ \&\&\ (i_3 <> \text{'!'})$

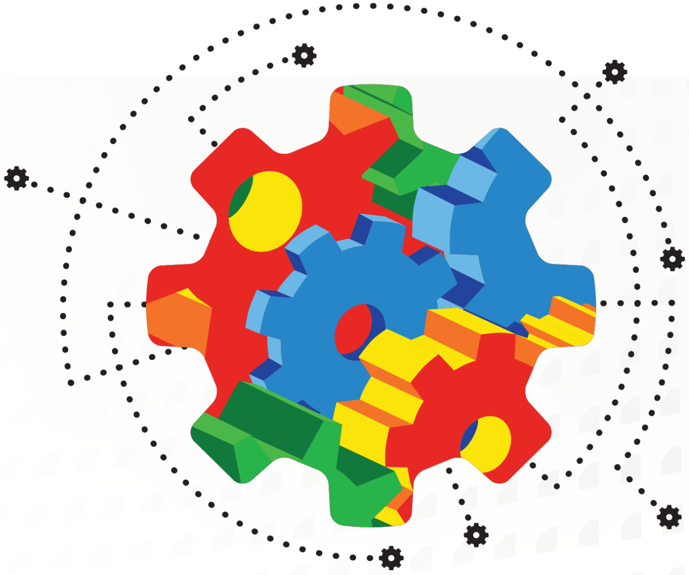
Does It Blend?

```
int main(int argc, char *argv[])
{
  char buf[500];
  size_t count;
  fd = open(argv[1], O_RDONLY);
  if(fd == -1) {
    perror("open");
    exit(-1);
  }
  count = read(fd, buf, 500);
  if(count == -1) {
    perror("read");
    exit(-1);
  }
  close(fd);
  test(buf);
  return 0;

}
```

```
void crash(){
    *(int*)NULL = 0;
}

void test(char * buf)
{
    int n=0;
    if(buf[0] == 'b') n++;
    if(buf[1] == 'a') n++;
    if(buf[2] == 'd') n++;
    if(buf[3] == '!') n++;
    if(n==4){
        crash();
    }
}
```

Does It Blend?

```
moflow@ubuntu:~/moflow-bap-0.7/custom_utils/egas$ ./egas -app test/bof1 -seed test/input.txt
Starting program
Thread 0 starting
Opening tainted file: samples/13.sol
Tainting 5 bytes from read at bffafe30
buffer_size: 5, requested length: 5
Taint introduction #0. @bffafe30/5 bytes: file samples/13.sol
adding new mapping from file samples/13.sol to 0 on taint num 1
adding new mapping from file samples/13.sol to 1 on taint num 2
adding new mapping from file samples/13.sol to 2 on taint num 3
adding new mapping from file samples/13.sol to 3 on taint num 4
adding new mapping from file samples/13.sol to 4 on taint num 5
Activating taint analysis
CRASH! Sample: samples/13.sol saved as crashes/2014-06-20_22:40:10_13.crash
----------STATS----------
%     total count desc
68%   13s   9     taint tracing the target (produces .bpt)
16%   3s    14    gathering coverage info
5%    1s    9     symbolic execution
0%    0s    0     .bpt concretization
0%    0s    13    solver interaction
11%   2s    1     unaccounted
-------------------------
elapsed: 19.000000
```



Microsoft SAGE

Implementation

Tester	• AppVerifier crash harness
Tracer	• iDNA DBI Framework
CoverageCollector	• Coverage analysis of iDNA trace using Nirvana
SymbolicExecutor	• X86->SMT translator and constraint collector over iDNA trace using TruScan
Disolver	• Z3 constraint solver

Optimizations

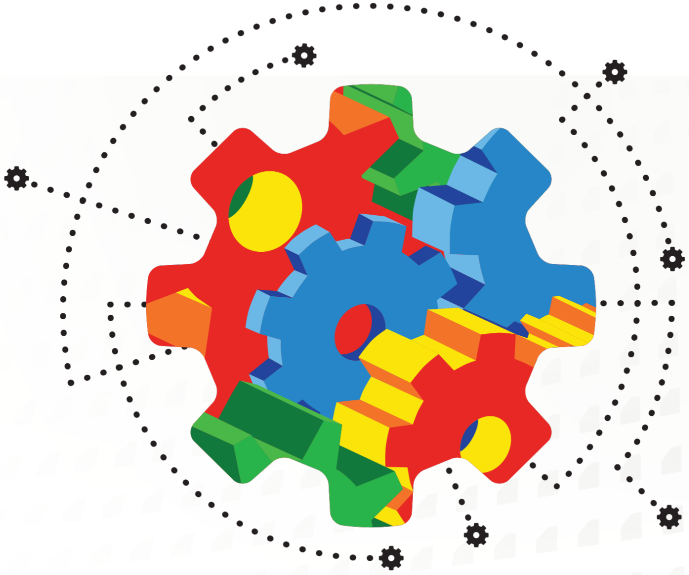
- Generational Search vs DFS
  - → DFS or BFS would negate only one of the branches
  - → Generational search negates each condition and solves for each, generating many new inputs per symbolic execution phase instead of just one

- Constraint Optimization
  - → Constraint Elimination - reduces the size of constraint solver queries by removing the constraints which do not share symbolic variables with the negated constraint
  - → Local constraint Caching - skips a constraint if it has already been added to the path constraint
  - → Flip count limit - establishes the maximum number of times a constraint generated from a particular program instruction can be flipped
  - → Constraint Subsumption - tracks constraints dominated by a specific branch, skips identical constraints generated from the same instruction location

Results

- Thousands of crashes found in the Windows 7 and Office products – 1/3 of all file fuzzing bugs 2007-2009

- Lessons Learned
  - → Vulnerabilities discovered are usually at shallow code depths
  - → Symbolic Execution state is limited so wrappers need to be developed for library code
  - → A small number of generations typically find the majority of vulnerabilities



Moflow::FuzzFlow

Sourcefire is now part of Cisco.

Implementation

Tracer	• Modified BAP pintool to collect memory dumps, coverage information, input stream names. Detects exceptions as well
Symbolic Executor	• Modification to BAP that supports converting BAP IL to SMTLIB formulas
SMT Solver	• We use z3 or STP to solve generated formulas
FuzzFlow Logic	• Custom tool built on top of BAP that glues all components together and implements the exploration algorithm

Limitations

- Tracer
  - → Taint tracer from BAP is not optimized
  - → Code depth and complexity constrained by input size
  - → PIN is unable to flush single basic block hooks from code cache for code coverage hit trace
- Symbolic Execution
  - → Slow conversion from BIL to SMTLIB on big traces
- FuzzFlow
  - → Libraries need to be wrapped directly
  - → We lack most of the optimizations in SAGE such as constraint subsumption

Real World Vulnerability Discovery

```
moflow@ubuntu:~/moflow-bap-0.7/custom_utils/egas$ ./egas -app /home/moflow/graphite2-1.2.3/
tests/comparerenderer/comparerenderer -seed /home/moflow/graphite2-1.2.3/tests/fonts/tiny.ttf -
fmt "-t /home/moflow/graphite2-1.2.3/tests/texts/udhr_nep.txt -s 12 -f %s -n"



Breakpoint 1, _IO_fread (buf=0x0, size=1, count=3758096384, fp=0x8053230) at iofread.c:37
37  in iofread.c
(gdb) bt
#0  _IO_fread (buf=0x0, size=1, count=3758096384, fp=0x8053230) at iofread.c:37
#1  0x4003a8ca in graphite2::FileFace::get_table_fn(void const*, unsigned int, unsigned int*) ()
   from /home/moflow/graphite2-1.2.3/src/libgraphite2.so.3
#2  0x4002e8e5 in graphite2::Face::Table::Table(graphite2::Face const&, graphite2::TtfUtil::Tag)
()
   from /home/moflow/graphite2-1.2.3/src/libgraphite2.so.3
#3  0x4002858a in (anonymous namespace)::load_face(graphite2::Face&, unsigned int) ()
   from /home/moflow/graphite2-1.2.3/src/libgraphite2.so.3
#4  0x40028695 in gr_make_face_with_ops () from /home/moflow/graphite2-1.2.3/src/
libgraphite2.so.3
#5  0x40028aac in gr_make_file_face () from /home/moflow/graphite2-1.2.3/src/libgraphite2.so.3
#6  0x0804d56d in Gr2Face::Gr2Face(char const*, int, std::string const&, bool) ()
#7  0x0804b664 in main ()
```

Real World Vulnerability Discovery

```
const void *FileFace::get_table_fn(const void* appFaceHandle, unsigned int name, size_t *len)
{
    if (appFaceHandle == 0) return 0;
    const FileFace & file_face = *static_cast<const FileFace *>(appFaceHandle);
    void *tbl;
    size_t tbl_offset, tbl_len;
    if (!TtfUtil::GetTableInfo(name, file_face._header_tbl,
                               file_face._table_dir, tbl_offset, tbl_len))
        return 0;

    if (tbl_offset + tbl_len > file_face._file_len
            || fseek(file_face._file, tbl_offset, SEEK_SET) != 0)
        return 0;

    tbl = malloc(tbl_len);
    if (fread(tbl, 1, tbl_len, file_face._file) != tbl_len)
    {
        free(tbl);
        return 0;
    }

    if (len) *len = tbl_len;
    return tbl;
}
```

Real World Vulnerability Discovery

- LZO / LZ4 vulnerability discovered by “In all variants of LZ[O4], the vulnerability occurs when processing a Literal Run. This is a chunk of compressed data that isn't compressed at all. Literals are uncompressed bytes that the user decided, for whatever reason, should not be compressed. A Literal Run is signaled by a state machine in LZO, and by a Mask in LZ4.”
  - Don Bailey June 26th, 2014

Real World Vulnerability Discovery

```
if (likely(state == 0)) {
        if (unlikely(t == 0)) {
                while (unlikely(*ip == 0)) {
                        t += 255;
                        ip++;
                        NEED_IP(1);
                }
                t += 15 + *ip++;
        }
        t += 3;
```

If the payload contains a NULL byte, ‘t’ is incremented by 255 leading to the ability to assign large 32-bit unsigned integers to ‘t’

Real World Vulnerability Discovery

```
66 copy_literal_run:
67 #if defined(CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS)
68                                  if (likely(HAVE_IP(t + 15) && HAVE_OP(t + 15))) {
69                                          const unsigned char *ie = ip + t;
70                                          unsigned char *oe = op + t;
71                                          do {
72                                                  COPY8(op, ip);
73                                                  op += 8;
74                                                  ip += 8;
75                                                  COPY8(op, ip);
76                                                  op += 8;
77                                                  ip += 8;
78                                          } while (ip < ie);
79                                          ip = ie;
80                                          op = oe;
81                                  } else
82 #endif
```

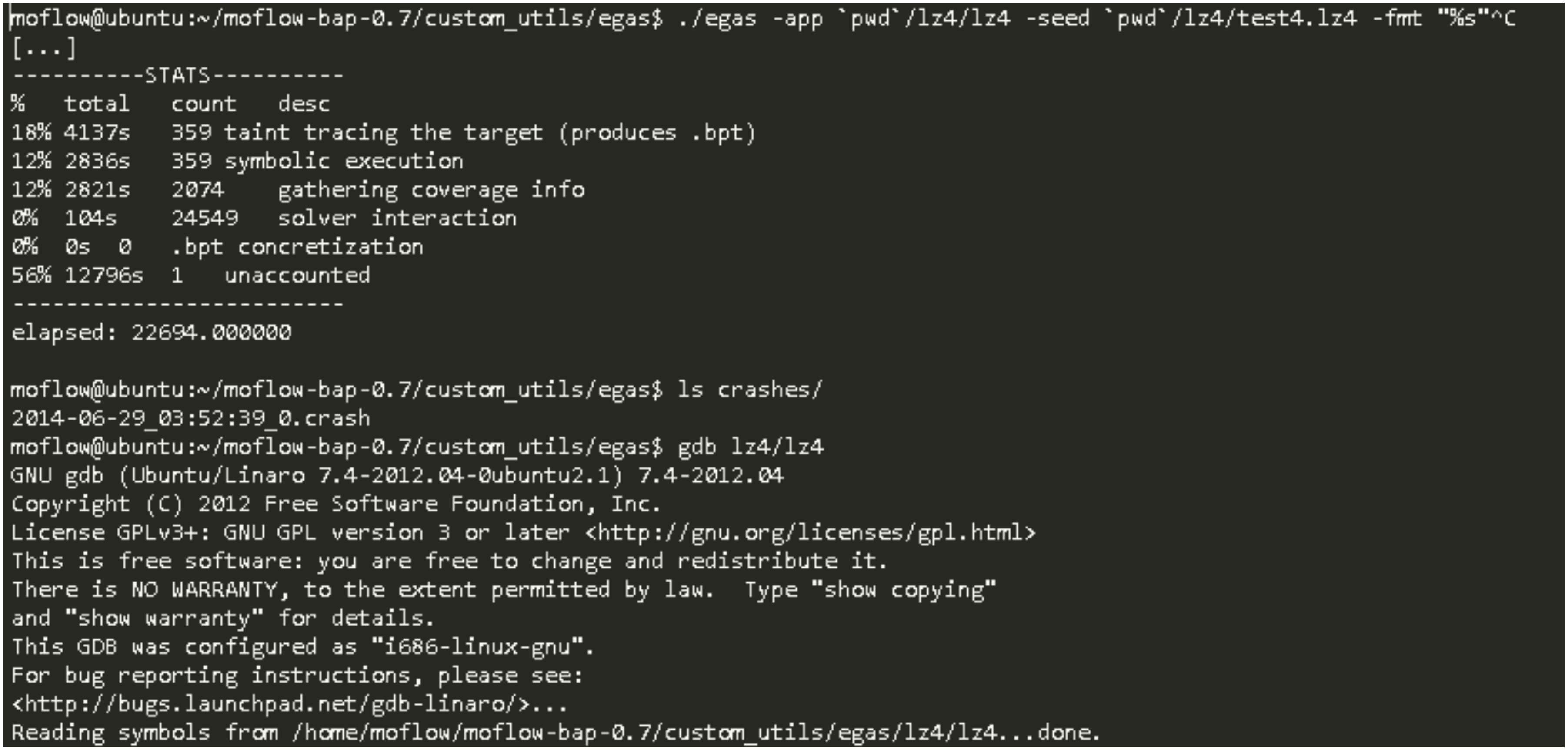
Line 68 validates the size of the src and dst buffers with ‘t + 15’ (t is size)

If ‘t’ is large the value will wrap the addition and the check will pass.

Loop will exit after 16 bytes

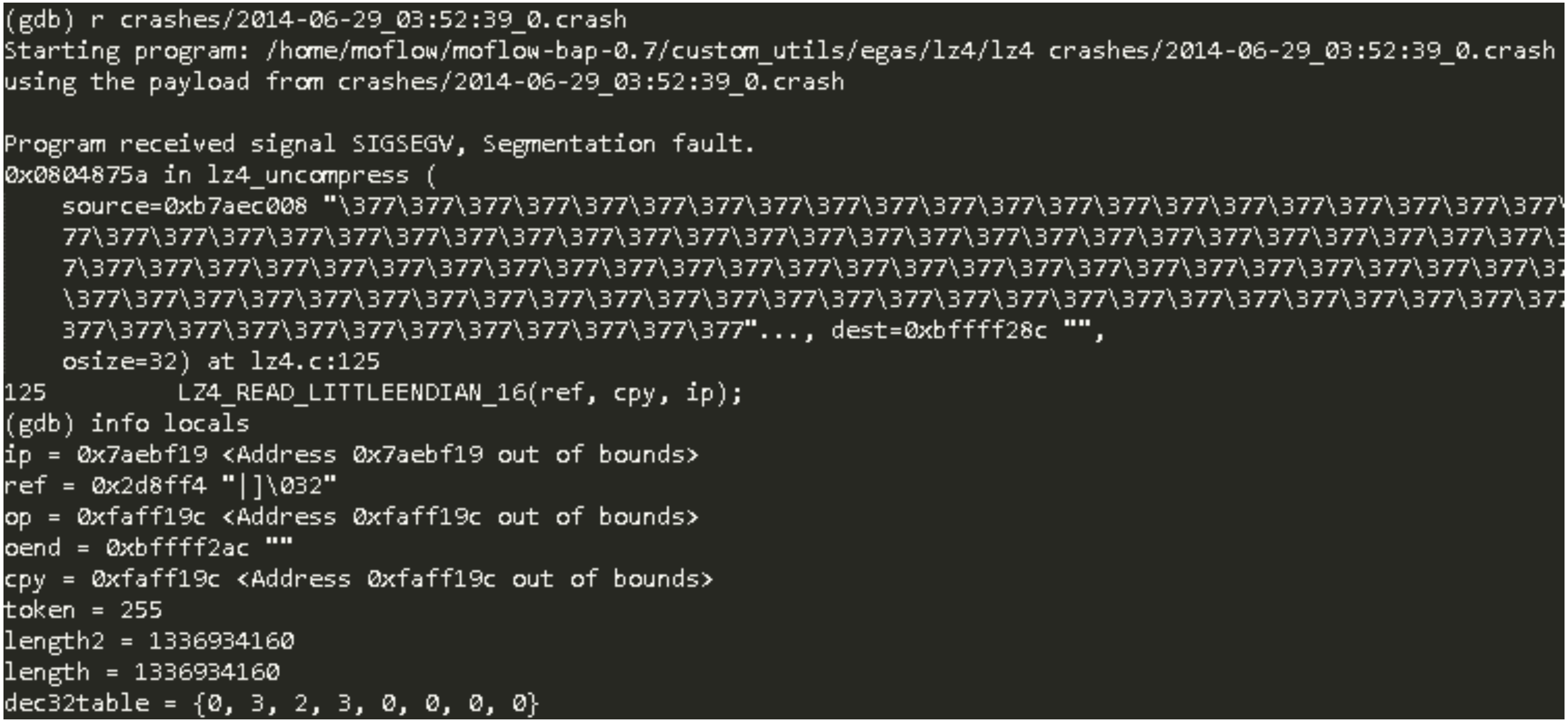
‘ip’ and ‘op’ will be set to invalid memory pointers

Real World Vulnerability Discovery

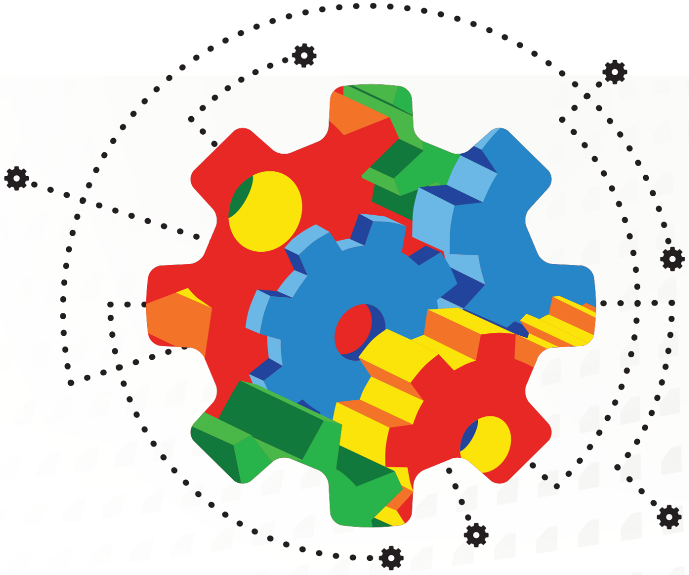


```
moflow@ubuntu:~/moflow-bap-0.7/custom_utils/egas$ ./egas -app `pwd`/lz4/lz4 -seed `pwd`/lz4/test4.lz4 -fmt "%s"^C
[...]
----------STATS----------
%   total   count   desc
18% 4137s   359 taint tracing the target (produces .bpt)
12% 2836s   359 symbolic execution
12% 2821s   2074    gathering coverage info
0%  104s    24549   solver interaction
0%  0s  0   .bpt concretization
56% 12796s  1   unaccounted
-------------------------
elapsed: 22694.000000

moflow@ubuntu:~/moflow-bap-0.7/custom_utils/egas$ ls crashes/
2014-06-29_03:52:39_0.crash
moflow@ubuntu:~/moflow-bap-0.7/custom_utils/egas$ gdb lz4/lz4
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/moflow/moflow-bap-0.7/custom_utils/egas/lz4/lz4...done.
```

Real World Vulnerability Discovery



```
(gdb) r crashes/2014-06-29_03:52:39_0.crash
Starting program: /home/moflow/moflow-bap-0.7/custom_utils/egas/lz4/lz4 crashes/2014-06-29_03:52:39_0.crash
using the payload from crashes/2014-06-29_03:52:39_0.crash

Program received signal SIGSEGV, Segmentation fault.
0x0804875a in lz4_uncompress (
    source=0xb7aec008 "\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\
    77\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\3
    7\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\3
    \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\37
    377\377\377\377\377\377\377\377\377\377\377\377"..., dest=0xbffff28c "",
    osize=32) at lz4.c:125
125         LZ4_READ_LITTLEENDIAN_16(ref, cpy, ip);
(gdb) info locals
ip = 0x7aebf19 <Address 0x7aebf19 out of bounds>
ref = 0x2d8ff4 "|]\032"
op = 0xfaff19c <Address 0xfaff19c out of bounds>
oend = 0xbffff2ac ""
cpy = 0xfaff19c <Address 0xfaff19c out of bounds>
token = 255
length2 = 1336934160
length = 1336934160
dec32table = {0, 3, 2, 3, 0, 0, 0, 0}
```



Binary Differencing

The Good Old Days

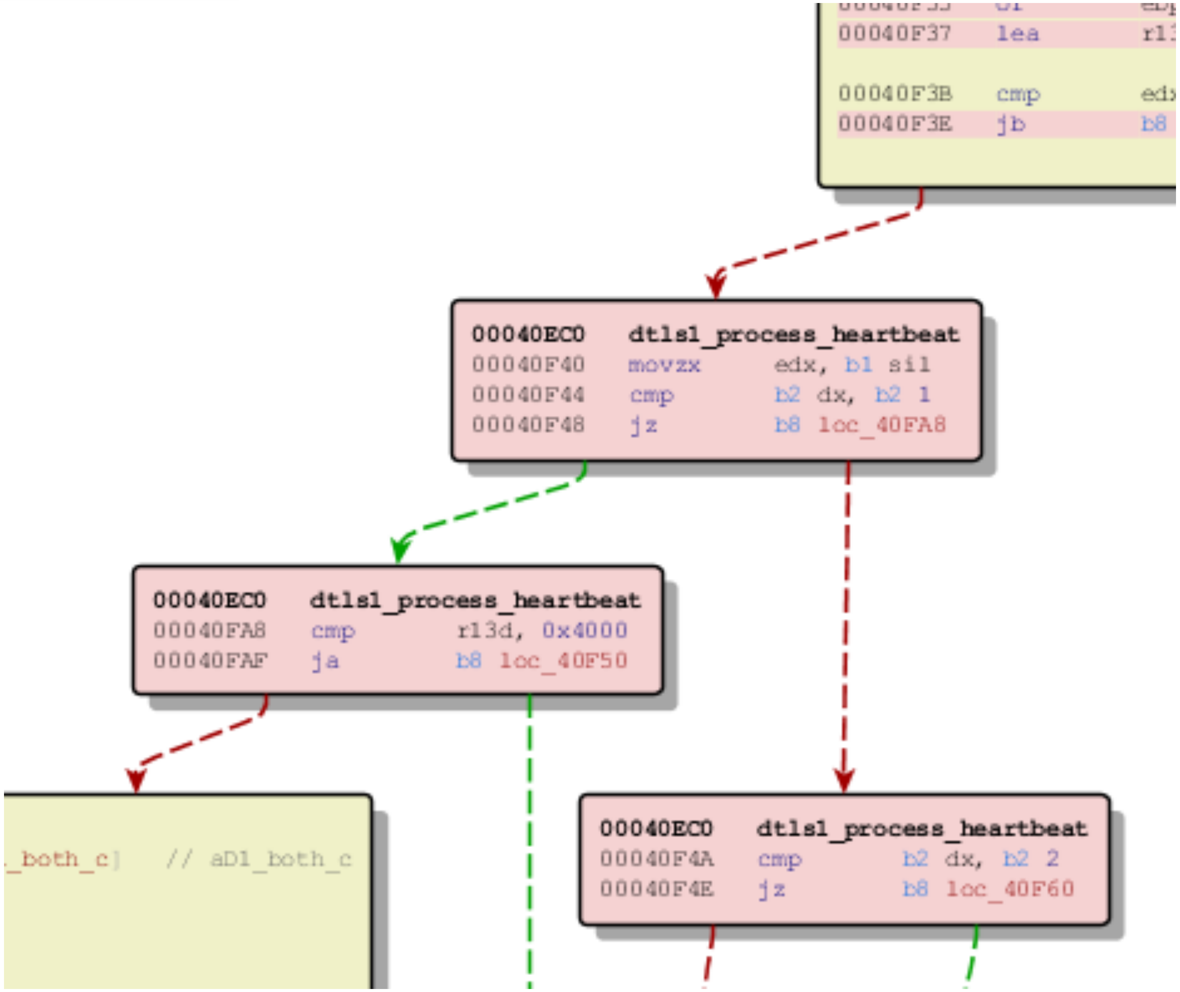
- In 2004, Halvar was the first to apply isomorphic graph comparison to the problem of binary program differencing
- The primary class of vulnerabilities at the time were integer overflows
  - → "Integer overflows are heavily represented in OS vendor advisories, rising to number 2 in 2006" http://cwe.mitre.org/documents/vuln-trends/index.html
  - → Integer Overflows are localized vulnerabilities that result in array indexing or heap allocation size miscalculations
- Many vulnerabilities were targeting file formats such a Microsoft Office

BinDiff in 2014

- Last update for the only commercialized BinDiff tool (Zynamics BinDiff) was in 2011

- Use-after-free bugs are king
  - → First added to CWE in 2008, UAF now dominates as a vulnerability class in web-browsers and document parsers
  - → High degree of separation between the root cause and trigger

Inline Bounds Checking



```
00040F37    lea         r1
00040F3B    cmp         ed
00040F3E    jb          b8
```

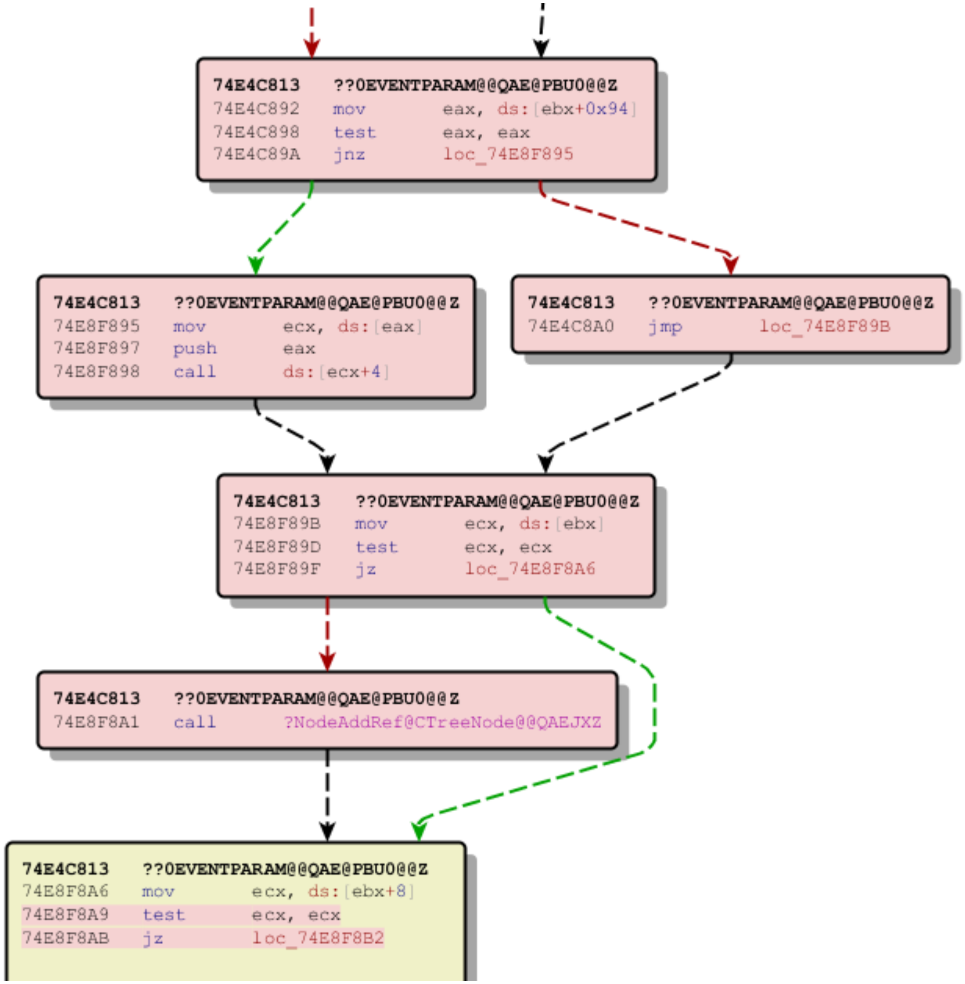
```
00040EC0    dtls1_process_heartbeat
00040F40    movzx       edx, b1 sil
00040F44    cmp         b2 dx, b2 1
00040F48    jz          b8 loc_40FA8
```

```
00040EC0    dtls1_process_heartbeat
00040FA8    cmp         r13d, 0x4000
00040FAF    ja          b8 loc_40F50
```

```
_both_c]    // aD1_both_c
```

```
00040EC0    dtls1_process_heartbeat
00040F4A    cmp         b2 dx, b2 2
00040F4E    jz          b8 loc_40F60
```

Use-After-Free



```
74E4C813    ??0EVENTPARAM@@QAE@PBU0@@Z
74E4C892    mov         eax, ds:[ebx+0x94]
74E4C898    test        eax, eax
74E4C89A    jnz         loc_74E8F895
```

```
74E4C813    ??0EVENTPARAM@@QAE@PBU0@@Z
74E8F895    mov         ecx, ds:[eax]
74E8F897    push        eax
74E8F898    call        ds:[ecx+4]
```

```
74E4C813    ??0EVENTPARAM@@QAE@PBU0@@Z
74E4C8A0    jmp         loc_74E8F89B
```

```
74E4C813    ??0EVENTPARAM@@QAE@PBU0@@Z
74E8F89B    mov         ecx, ds:[ebx]
74E8F89D    test        ecx, ecx
74E8F89F    jz          loc_74E8F8A6
```

```
74E4C813    ??0EVENTPARAM@@QAE@PBU0@@Z
74E8F8A1    call        ?NodeAddRef@CTreeNode@@QAEJXZ
```

```
74E4C813    ??0EVENTPARAM@@QAE@PBU0@@Z
74E8F8A6    mov         ecx, ds:[ebx+8]
74E8F8A9    test        ecx, ecx
74E8F8AB    jz          loc_74E8F8B2
```

Function Matching

- Hash Matching (bytes/names)
- MD index matching (flowgraph hash)
- Instruction count
- Address sequence
- String references
- Loop count
- Call sequence

Basic Block Matching

- Edges Prime Product
- Hash/Prime
- MD index (flowgraph hash)
- Loop entry
- Entry/Exit point
- Jump sequence

Practical Problems

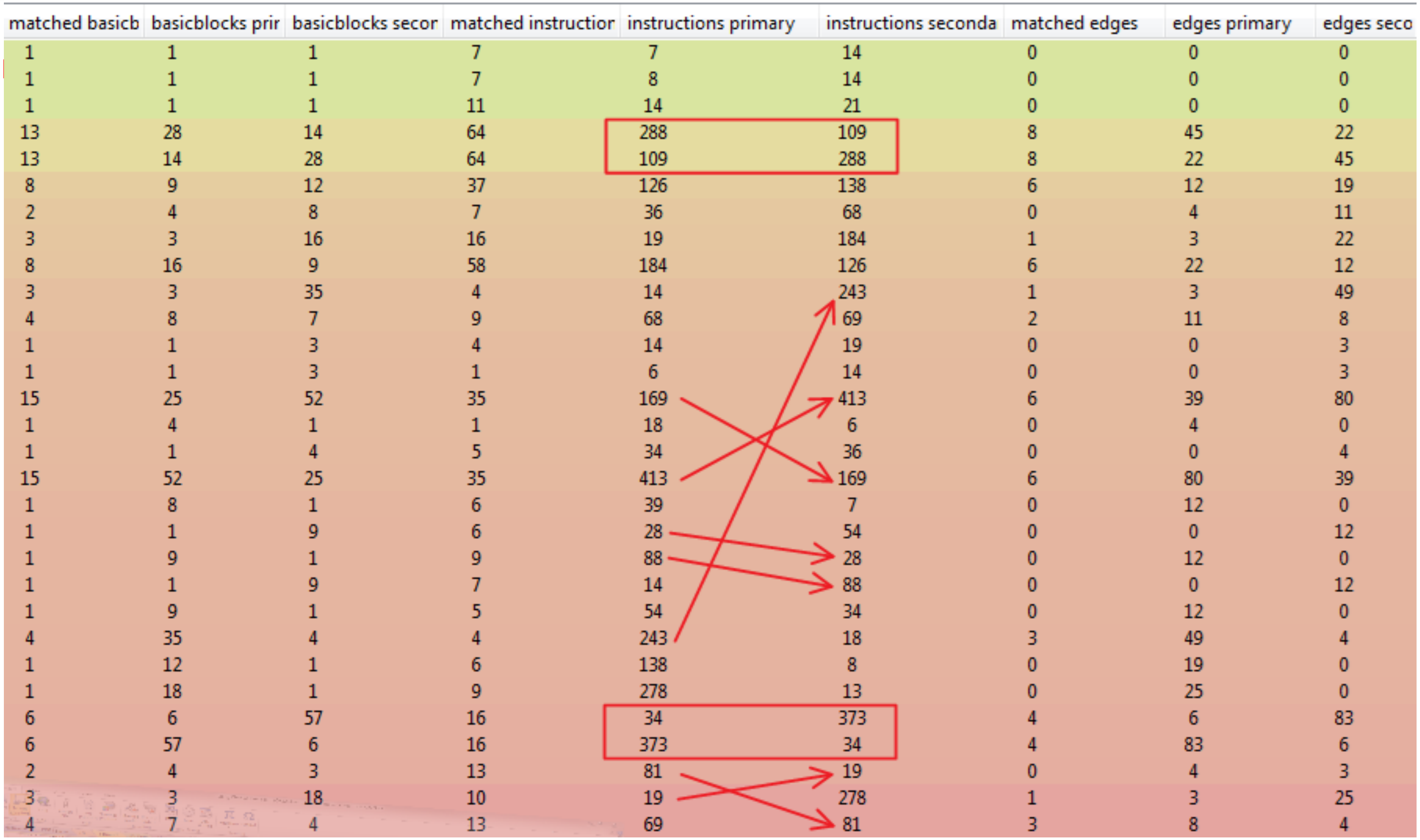
- Mismatched functions
  - → Some functions are identical in both binaries, but mismatched by the differ
- Assembly refactoring
  - → Some functions are semantically identical in both binaries, but some assembly instructions have changed/ moved
- Little to no context
  - → Functions are given a similarity rating, but no potential indicators of security-related additions

Practical Problems

- Compiler optimizations are not handled
- Chunked functions are not handled
- BinDiff heuristics are not tunable / configurable
- IDA misidentifies data as code

- UAF vulnerabilities are hard to reverse engineer
  - → The DOM is massive and interactions between objects are not defined
  - → The patches are typically simple reference counting patches (add missing calls to AddRef)

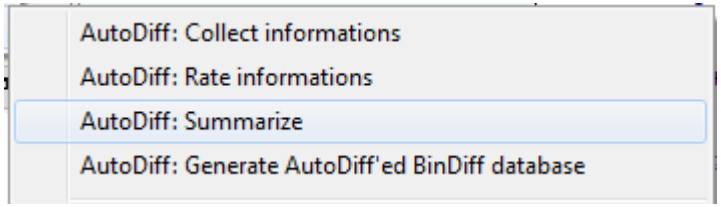
Mismatched Functions



matched basicb	basicblocks prir	basicblocks secor	matched instructior	instructions primary	instructions seconda	matched edges	edges primary	edges seco
1	1	1	7	7	14	0	0	0
1	1	1	7	8	14	0	0	0
1	1	1	11	14	21	0	0	0
13	28	14	64	288	109	8	45	22
13	14	28	64	109	288	8	22	45
8	9	12	37	126	138	6	12	19
2	4	8	7	36	68	0	4	11
3	3	16	16	19	184	1	3	22
8	16	9	58	184	126	6	22	12
3	3	35	4	14	243	1	3	49
4	8	7	9	68	69	2	11	8
1	1	3	4	14	19	0	0	3
1	1	3	1	6	14	0	0	3
15	25	52	35	169	413	6	39	80
1	4	1	1	18	6	0	4	0
1	1	4	5	34	36	0	0	4
15	52	25	35	413	169	6	80	39
1	8	1	6	39	7	0	12	0
1	1	9	6	28	54	0	0	12
1	9	1	9	88	28	0	12	0
1	1	9	7	14	88	0	0	12
1	9	1	5	54	34	0	12	0
4	35	4	4	243	18	3	49	4
1	12	1	6	138	8	0	19	0
1	18	1	9	278	13	0	25	0
6	6	57	16	34	373	4	6	83
6	57	6	16	373	34	4	83	6
2	4	3	13	81	19	0	4	3
3	3	18	10	19	278	1	3	25
4	7	4	13	69	81	3	8	4

AutoDiff

- Our solution is to post-process the database generated from BinDiff
- We augment the existing database by performing further analysis with IDApython scripts
- New tables are added to supplement the existing information


AutoDiff: Collect informations
AutoDiff: Rate informations
AutoDiff: Summarize
AutoDiff: Generate AutoDiff'ed BinDiff database

AutoDiff

- Features
  - → Instruction counting (including chunked function)
  - → Instructions added/removed from each function
  - → IntSafe library awareness
  - → Filtering of innocuous / superfluous changes
  - → Filtering of changes without a security impact
    - Example: new 'ret' instructions generated by compiler
  - → Mnemonic list comparison
    - To determine when register substitution is the only change

Results

- MS13-097 – ieinstal.dll: 19% reduction

```
=======================================================
=                AutoDiff / Statistics                =
=======================================================
Number of changed functions declared by BinDiff : 179
Number of functions filtered out by Sanitizer   : 26
Number of functions contain "IntSafe patch"     : 1
Number of functions ReMatched                   : 7
Number of functions still left to analysis      : 145
```

Results

- MS14-017 – wordcnv.dll: 76% reduction

```
=======================================================
=                AutoDiff / Statistics                =
=======================================================
Number of changed functions declared by BinDiff : 55
Number of functions filtered out by Sanitizer   : 0
Number of functions contain "IntSafe patch"     : 0
Number of functions ReMatched                   : 42
Number of functions still left to analysis      : 13
```

Results

- MS14-035 – urlmon.dll: 29% reduction

```
=======================================================
=                AutoDiff / Statistics                =
=======================================================
Number of changed functions declared by BinDiff : 31
Number of functions filtered out by Sanitizer   : 9
Number of functions contain "IntSafe patch"     : 0
Number of functions ReMatched                   : 0
Number of functions still left to analysis      : 22
```

Results

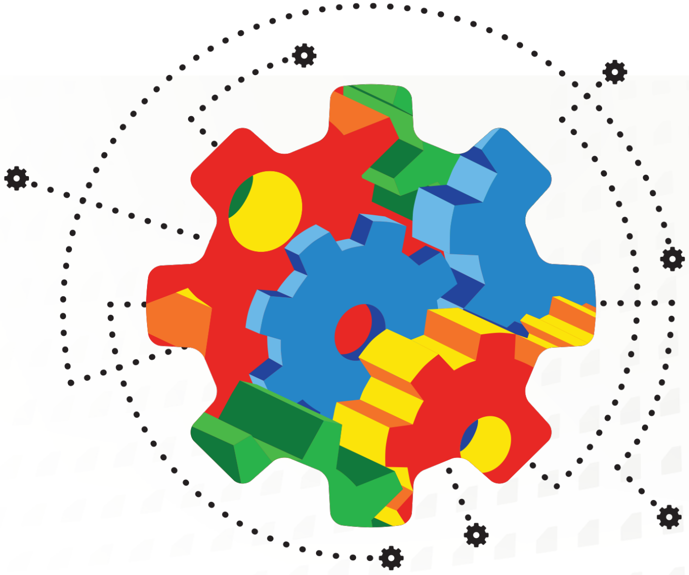
- MS14-035 – mshtml.dll: 21% reduction

```
=======================================================
=                AutoDiff / Statistics                =
=======================================================
Number of changed functions declared by BinDiff : 543
Number of functions filtered out by Sanitizer   : 56
Number of functions contain "IntSafe patch"     : 0
Number of functions ReMatched                   : 61
Number of functions still left to analysis      : 426
```

Results

- Adobe CVE-2014-0497: 87% reduction

```
=======================================================
=                AutoDiff / Statistics                =
=======================================================
Number of changed functions declared by BinDiff : 1118
Number of functions filtered out by Sanitizer   : 975
Number of functions contain "IntSafe patch"     : 0
Number of functions ReMatched                   : 0
Number of functions still left to analysis      : 143
```



Semantic Difference Engine

BinDiff Problem Areas

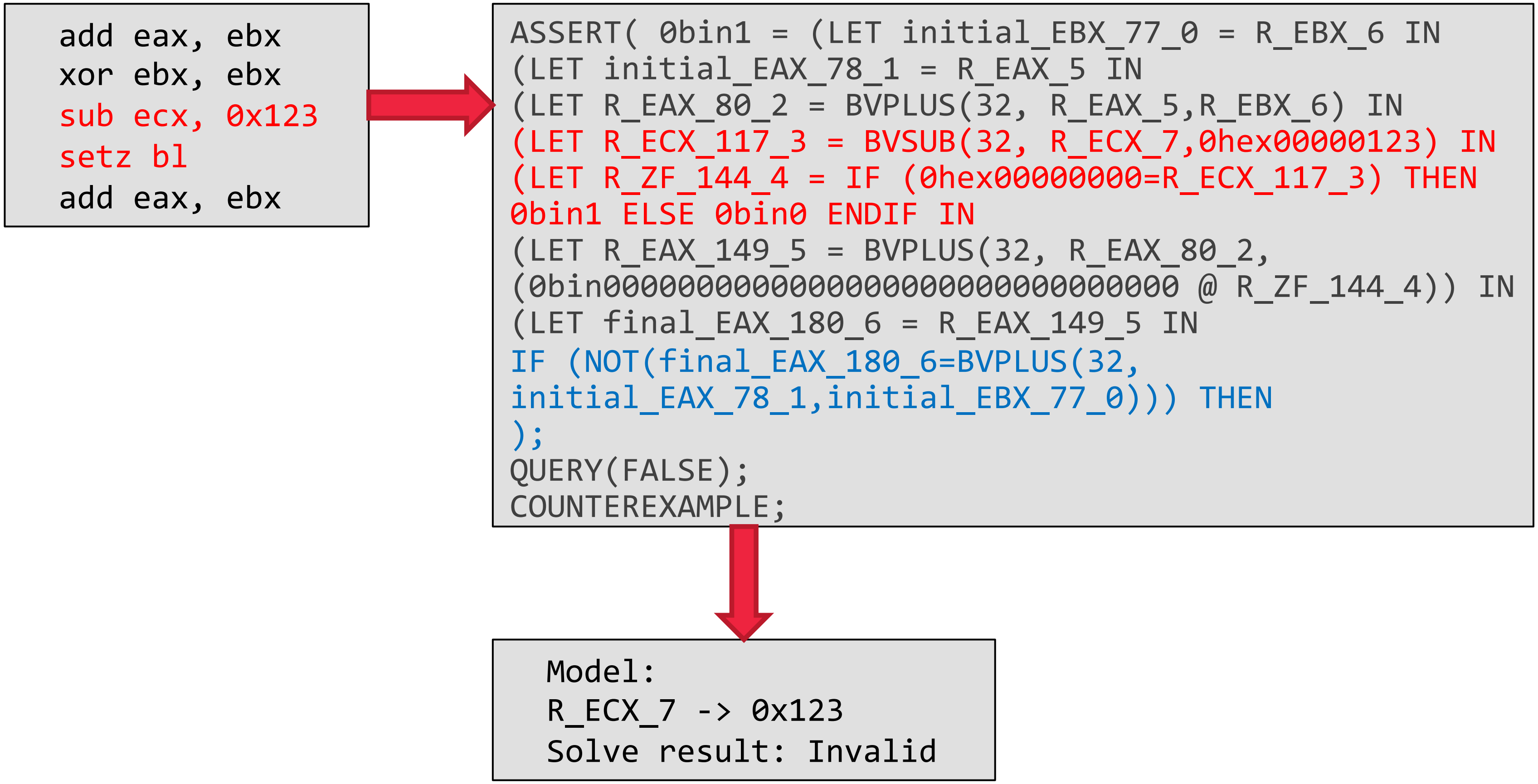
- Reassignment of registers while maintaining the same semantics
- Inversion of branch logic
  - → such as `jge -> jl`
- Using more optimized assembler instructions that are semantically equivalent

The Idea

- We've shown success using symbolic execution to analyze code paths to generate inputs
- We should be able to ask a solver to tell us if two sets of code are equivalent
- In last year's presentation we showed an example of exactly this
  - → Is "`add eax, ebx`" equivalent to this code:

```
add eax, ebx
xor ebx, ebx
sub ecx, 0x123
setz bl
add eax, ebx
```

The Idea



```
add eax, ebx
xor ebx, ebx
sub ecx, 0x123
setz bl
add eax, ebx
```

```
ASSERT( 0bin1 = (LET initial_EBX_77_0 = R_EBX_6 IN
(LET initial_EAX_78_1 = R_EAX_5 IN
(LET R_EAX_80_2 = BVPLUS(32, R_EAX_5,R_EBX_6) IN
(LET R_ECX_117_3 = BVSUB(32, R_ECX_7,0hex00000123) IN
(LET R_ZF_144_4 = IF (0hex00000000=R_ECX_117_3) THEN
0bin1 ELSE 0bin0 ENDIF IN
(LET R_EAX_149_5 = BVPLUS(32, R_EAX_80_2,
(0bin0000000000000000000000000000000 @ R_ZF_144_4)) IN
(LET final_EAX_180_6 = R_EAX_149_5 IN
IF (NOT(final_EAX_180_6=BVPLUS(32,
initial_EAX_78_1,initial_EBX_77_0))) THEN
);
QUERY(FALSE);
COUNTEREXAMPLE;
```

```
Model:
R_ECX_7 -> 0x123
Solve result: Invalid
```

The Idea

- Strategy would be to mark function parameters as symbolic and discover each path constraint to solve for inputs that would reach all paths
- At termination of each path the resulting CPU state and variable values should be identical
- Unfortunately this led to a false impression of the feasibility of this approach

The Reality

- Low level IR is tied to a memory and register model
- This level of abstraction does not sufficiently alias references to the same memory
- At minimum private symbol information would be needed to abstract beyond the memory addresses so we could manually match the values
- Decompilation would be a better first step towards this strategy, but symbol names are not guaranteed to match

A Practical Approach

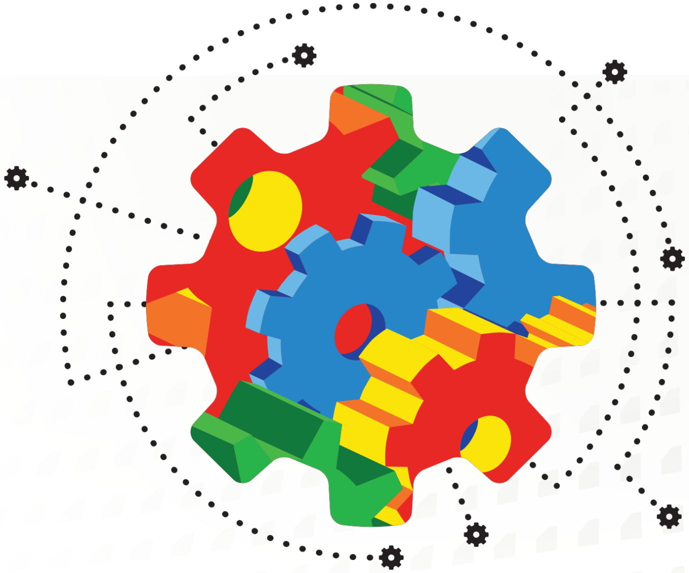
- David Ramos and Dawson Engler published "Practical, low-effort equivalence verification of real code" which shows a technique for performing a semantic equivalence test against source code using a modified version of KLEE
- Original application was for program verification of new implementations vs reference implementations, our problem is a subset of this
- Turns out the approach is nearly identical but works on a higher level of abstraction

A Practical Approach

- Code is compiled with symbol information using KLEE/LLVM
- A test harness is linked against each of the two functions to be compared
- The harness marks each parameter of the two functions as symbolic
- If input parameters are dereferenced as pointers, memory is lazily allocated as symbolic values
- Symbolically executes each function for each discovered constraint
- At the end of execution, KLEE traverses each memory location and solves for equivalent values at each location
- On failure of this check, a concrete input is generated that can prove the functions are different, else they've been proven equal

Where to Next

- The ability to alias memory references through the use of symbol information is the crucial missing piece of the puzzle for our approach
- There are additional difficulties with reference tracking, object comparison for passed parameters or return values, as well as overlapping memory references
- They explicitly specify that inline assembler is not handled due to their reliance on symbol information



Conclusions

Thank You!

- Cisco Talos VulnDev Team
  - → Richard Johnson
    - rjohnson@sourcefire.com
    - @richinseattle
  - → Ryan Pentney
  - → Marcin Noga
  - → Yves Younan
  - → Piotr Bania
  - → Pawel Janic (emeritus)

  - → Code released!
    - https://github.com/vrtadmin/moflow