VERIBIN: Adaptive Verification of Patches at the Binary Level

Hongwei Wu*, Jianliang Wu†, Ruoyu Wu*, Ayushi Sharma*, Aravind Machiry* and Antonio Bianchi*
*Purdue University
{wu1685, wu1377, sharm616, amachiry, antoniob}@purdue.edu
†Simon Fraser University
wujl@sfu.ca

Motivation

- Safe to Apply (StA)
  - Does not introduce any modification that could potentially break the functionality of the original binary.
  - *break functionality?
    - Does not increase valid input space of the binary
    - Output of the patched binary remains the same as the original for all valid inputs
- Given an original and patched binary, VeriBin uses symbolic execution to identify if the patch is StA or not.

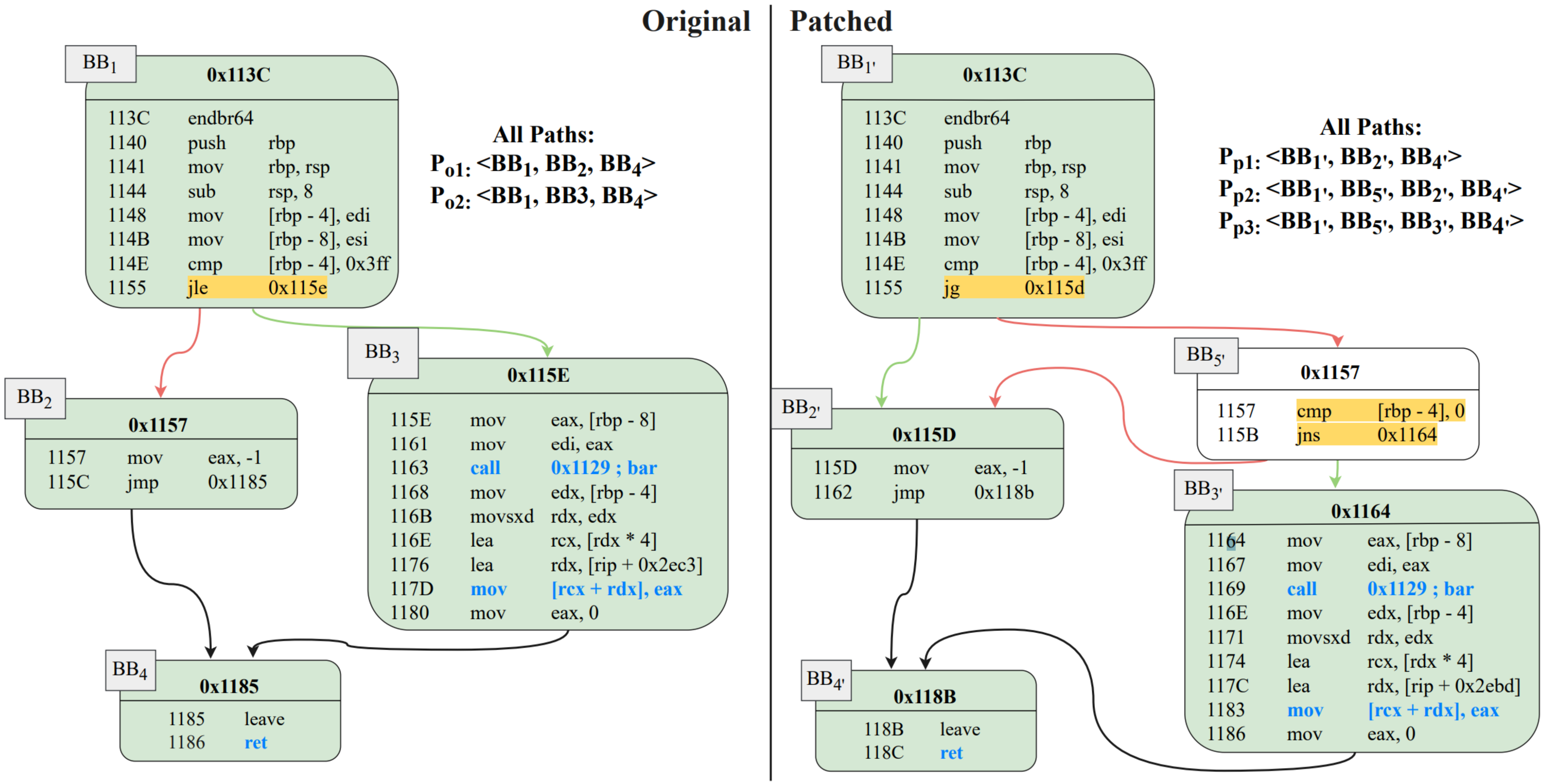
Scope

- StA
  - Does not increase input space
  - Does not affect program behavior for all valid inputs
- Many types of Patches
  - “Security Patches”
    - Usually only restricts input

Security Patch

```
#define MAX_SIZE 1024
int data[1024];
int foo(int a, int b){
-    if(a >= MAX_SIZE){
+    if(a >= MAX_SIZE || a < 0){
         return -1; }
     data[a] = bar(b);
     return 0; }
```

Listing 1: Running example



BB_1 0x113C
```
113C   endbr64
1140   push     rbp
1141   mov      rbp, rsp
1144   sub      rsp, 8
1148   mov      [rbp - 4], edi
114B   mov      [rbp - 8], esi
114E   cmp      [rbp - 4], 0x3ff
1155   jle      0x115e
```

BB_2 0x1157
```
1157   mov      eax, -1
115C   jmp      0x1185
```

BB_3 0x115E
```
115E   mov      eax, [rbp - 8]
1161   mov      edi, eax
1163   call     0x1129 ; bar
1168   mov      edx, [rbp - 4]
116B   movsxd   rdx, edx
116E   lea      rcx, [rdx * 4]
1176   lea      rdx, [rip + 0x2ec3]
117D   mov      [rcx + rdx], eax
1180   mov      eax, 0
```

BB_4 0x1185
```
1185   leave
1186   ret
```

All Paths:
P_{o1}: $<BB_1, BB_2, BB_4>$
P_{o2}: $<BB_1, BB3, BB_4>$

$BB_{1'}$ 0x113C
```
113C   endbr64
1140   push     rbp
1141   mov      rbp, rsp
1144   sub      rsp, 8
1148   mov      [rbp - 4], edi
114B   mov      [rbp - 8], esi
114E   cmp      [rbp - 4], 0x3ff
1155   jg       0x115d
```

$BB_{2'}$ 0x115D
```
115D   mov      eax, -1
1162   jmp      0x118b
```

$BB_{5'}$ 0x1157
```
1157   cmp      [rbp - 4], 0
115B   jns      0x1164
```

$BB_{3'}$ 0x1164
```
1164   mov      eax, [rbp - 8]
1167   mov      edi, eax
1169   call     0x1129 ; bar
116E   mov      edx, [rbp - 4]
1171   movsxd   rdx, edx
1174   lea      rcx, [rdx * 4]
117C   lea      rdx, [rip + 0x2ebd]
1183   mov      [rcx + rdx], eax
1186   mov      eax, 0
```

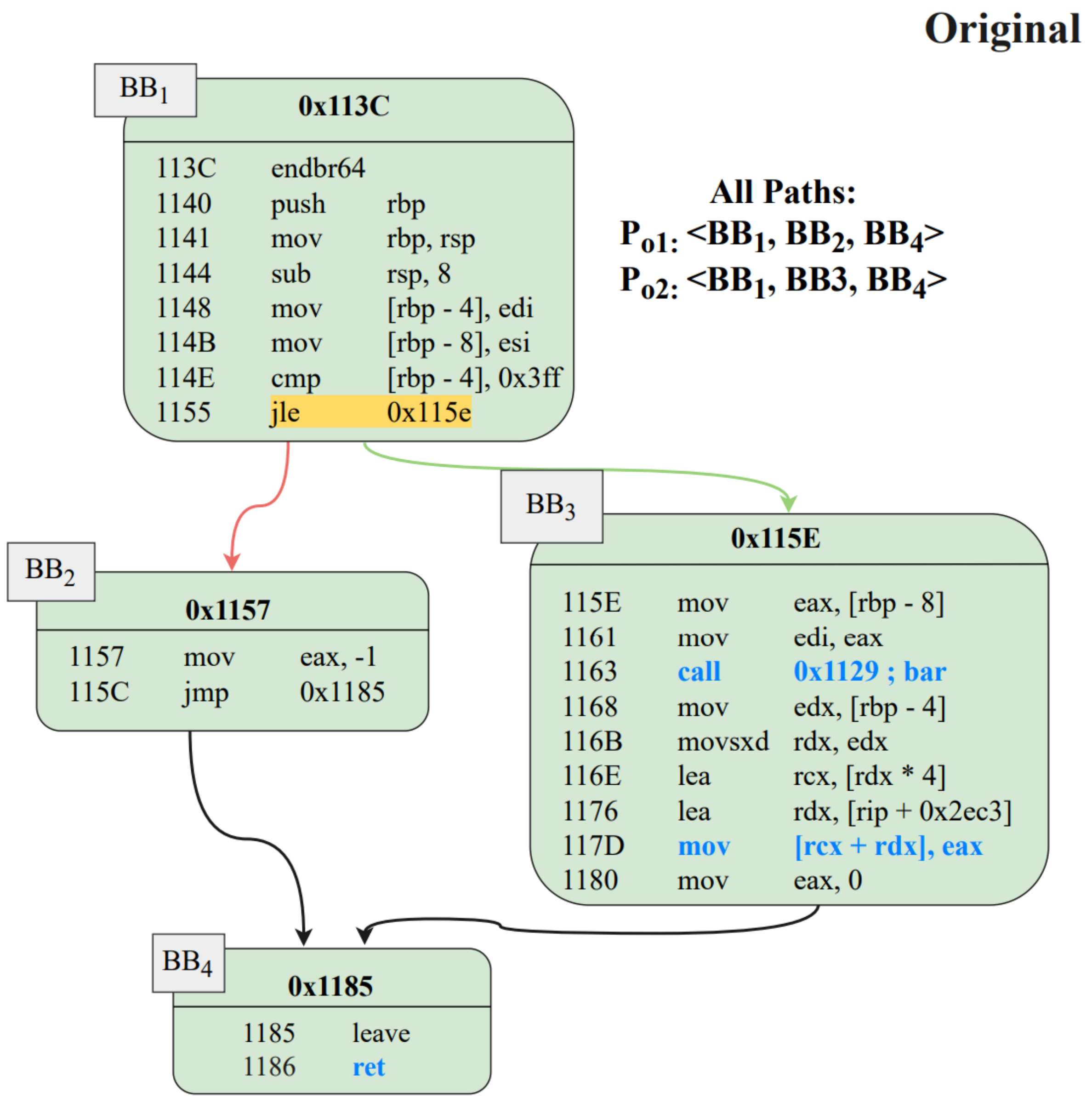
$BB_{4'}$ 0x118B
```
118B   leave
118C   ret
```

All Paths:
P_{p1}: $<BB_{1'}, BB_{2'}, BB_{4'}>$
P_{p2}: $<BB_{1'}, BB_{5'}, BB_{2'}, BB_{4'}>$
P_{p3}: $<BB_{1'}, BB_{5'}, BB_{3'}, BB_{4'}>$

Fig. 1: The Control Flow Graph (CFG) of the original and patched function in Listing 1. The instructions marked with blue denote the output of the function. The green basic blocks are matched blocks while the white basic block indicates the unmatched block. If a basic block has multiple out-going edges, we use the green and red edges to represent the true and false branches, respectively.

Terminology

- Valid Exit Path (VEP)
- Error-handling Exit Path (EEP)
- Path Constraint (PC)



All Paths:
P_{o1}: <BB_1, BB_2, BB_4>
P_{o2}: <BB_1, BB3, BB_4>

0x113C
```
113C    endbr64
1140    push    rbp
1141    mov     rbp, rsp
1144    sub     rsp, 8
1148    mov     [rbp - 4], edi
114B    mov     [rbp - 8], esi
114E    cmp     [rbp - 4], 0x3ff
1155    jle     0x115e
```

0x1157
```
1157    mov     eax, -1
115C    jmp     0x1185
```

0x115E
```
115E    mov     eax, [rbp - 8]
1161    mov     edi, eax
1163    call    0x1129 ; bar
1168    mov     edx, [rbp - 4]
116B    movsxd  rdx, edx
116E    lea     rcx, [rdx * 4]
1176    lea     rdx, [rip + 0x2ec3]
117D    mov     [rcx + rdx], eax
1180    mov     eax, 0
```

0x1185
```
1185    leave
1186    ret
```

Key Idea of Identifying StA

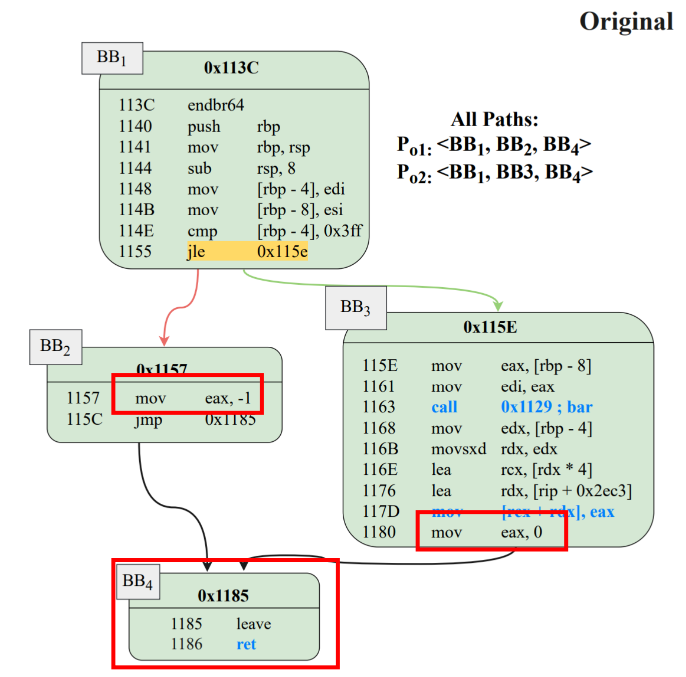
- Identify all EEPs
- All paths that is not a EEP is considered an VEP
- For every VEP, verify the StA properties

StA Properties

- P1: Non-Increasing Input Space
  - A valid input for VEP_p is also valid for VEP_o
- P2: Non-local Memory Writes
  - All non-local memory write operations on VEP_p should write the same value to the same memory region as the corresponding one in VEP_o
- P3: Return Value Equivalence
  - Return value along VEP_p should be the same as the return value of VEP_o
- P4: Function Call Equivalence
  - Function calls made along VEP_p should be equivalent to functions calls made along VEP_o

Challenges (Finding EEP)

- Application-specific error-handling functions
  - BUG, WARN in Linux Kernel
  - Often unavailable
- Error Labels
  - goto fatal
  - Often unavailable
- Return Value
  - Valid: return 0; Error: return -1
  - Compiler optimization combine return points
- VeriBin
  - Collect all executable paths + Symbolic Execution
  - Verify through *heuristics*


Original
All Paths:
P_{o1}: <BB_1, BB_2, BB_4>
P_{o2}: <BB_1, BB3, BB_4>

BB_1 0x113C
```
113C    endbr64
1140    push    rbp
1141    mov     rbp, rsp
1144    sub     rsp, 8
1148    mov     [rbp - 4], edi
114B    mov     [rbp - 8], esi
114E    cmp     [rbp - 4], 0x3ff
1155    jle     0x115e
```

BB_2 0x1157
```
1157    mov     eax, -1
115C    jmp     0x1185
```

BB_3 0x115E
```
115E    mov     eax, [rbp - 8]
1161    mov     edi, eax
1163    call    0x1129 ; bar
1168    mov     edx, [rbp - 4]
116B    movsxd  rdx, edx
116E    lea     rcx, [rdx * 4]
1176    lea     rdx, [rip + 0x2ec3]
117D    mov     [rcx + rdx], eax
1180    mov     eax, 0
```

BB_4 0x1185
```
1185    leave
1186    ret
```

Heuristics for EEP

- Call to Error-Handling Functions
    - exit, __assert_fail, abort … (dynamically linked)
    - FLIRT (statically linked)
- Return value
    - Invalid return values (e.g., -1)
        - Collect all return values
        - 2 unique (value associated with shortest execution path is invalid)
        - More than 2 (negative values are considered invalid)
        - Incorporate invalid return values specified by the analyst
- Length of execution path
    - EEP is relatively short compared to VEP
        - EEP if path length is less than 0.8 of the average length of all paths in the CFG.

Challenges (Handling Complex Symbolic Constraints)

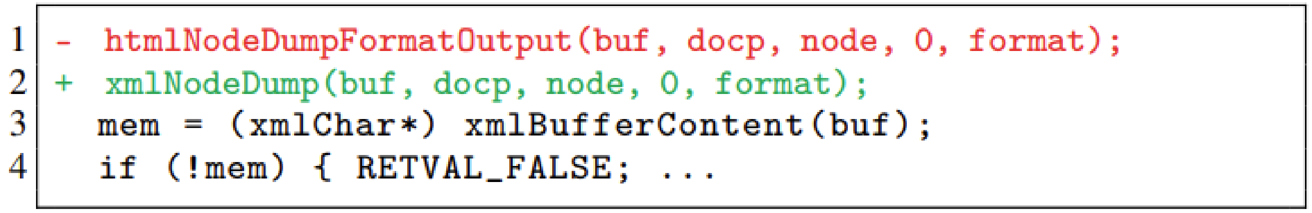
- Solving Path Constraints using SMT solver is challenging!
  - No Type information (Integer theory -> bit-vector theory)
  - Absence of variable names + other identifiers

- Matching Path Pairs (MPP)
  - Pair of VEP (one in patched and one in original)
  - Any input executed in VEP_p executes it VEP_o

Challenges (Handling Semantically Equivalent Changes)

- What if a function call is replaced with other equivalent function calls? Or what if we add calls to logging functions?
  - Does this violate P4? (Function call equivalence)

- Adaptive Verification Technique
  - Generate short prompts that analyst can answer to resolve these problems



```
- htmlNodeDumpFormatOutput(buf, docp, node, 0, format);
+ xmlNodeDump(buf, docp, node, 0, format);
  mem = (xmlChar*) xmlBufferContent(buf);
  if (!mem) { RETVAL_FALSE; ...
```

Listing 2: Example of a real-world patch that replaces a function call with an equivalent one

Adaptive Verification

- Q1 During P2 (Non-local Memory Write), when the original value and the patched value differ.
- Q2 When there are additional global writes in patched version
- Q3 When there are unmatched function calls
- Q4 When a function is called a different number of times
- Q5 When an argument in a matching function call pair is different

TABLE I: VERIBIN questions for adaptive verification. *Q*: type of questions. *Scenario*: specific conditions under which the question is asked. *Prompt*: the question or task presented.

Q	Scenario	Prompt
Q1	$\exists\langle a'_o, v'_o\rangle \in G_O,\ \langle a'_p, v'_p\rangle \in G_P$: $a'_o \equiv a'_p \wedge v'_o \not\equiv v'_p$	Can the difference between v'_o and v'_p be considered equivalent (StA)
Q2	$\forall\langle a_o, v_o\rangle \in G_O$, $\exists\langle a'_p, v'_p\rangle \in G_P$: $a_o \not\equiv a'_p$, or vice versa ($\langle a'_o, v'_o\rangle$)	Can VERIBIN consider $\langle a'_p, v'_p\rangle$ (or $\langle a'_o, v'_o\rangle$) to have no side effects (StA)?
Q3	$\forall cf_o \in VEP_o$, $\exists cf'_p \in VEP_p$: $cf_o \not\equiv cf'_p \wedge \forall cf_p \in VEP_p$, $\exists cf'_o \in VEP_o$: $cf_p \not\equiv cf'_o$	Can cf'_o and cf'_p be considered equivalent? ($cf'_o \equiv cf'_p$)
Q4	$\exists cf'_o \equiv cf'_p$: $N^o_{cf'_o} \neq N^p_{cf'_p}$	Can VERIBIN consider cf'_o (same as cf'_p) to have no side effects (StA)?
Q5	$\exists cf'_o \equiv cf'_p \wedge N^o_{cf'_o} = N^p_{cf'_p}$: $args^i_o \not\equiv args^i_p$	Can VERIBIN consider $args^i_o$ and $args^i_p$ to be equivalent?

Evaluation

TABLE III: Dataset Composition. *Arch.*: architecture is not supported. *No Func.*: Cannot detect patch-affected target functions. *Timeout*: Analysis timeout. *Memory*: Analysis memory exceeded.

Dataset	Total	Support	Excluded and Corresponding Reasons			
			Arch.	No Func	Timeout	Memory
D1	62	42	0	4	2	14
D2	7	6	1	0	0	0
D3	56	38	0	0	11	7
Unstripped	125	86		39		
D1_s	62	41	0	5	2	14
D2_s	7	6	1	0	0	0
D3_s	56	38	0	0	11	7
Stripped	125	85		40		

TABLE II: Experiments Description

	Adaptive	MPP	CIOC
E1	✗	✓	✓
E2	✓	✓	✓
E3	✗	✗	✓
E4	✗	✓	✗

Evaluation

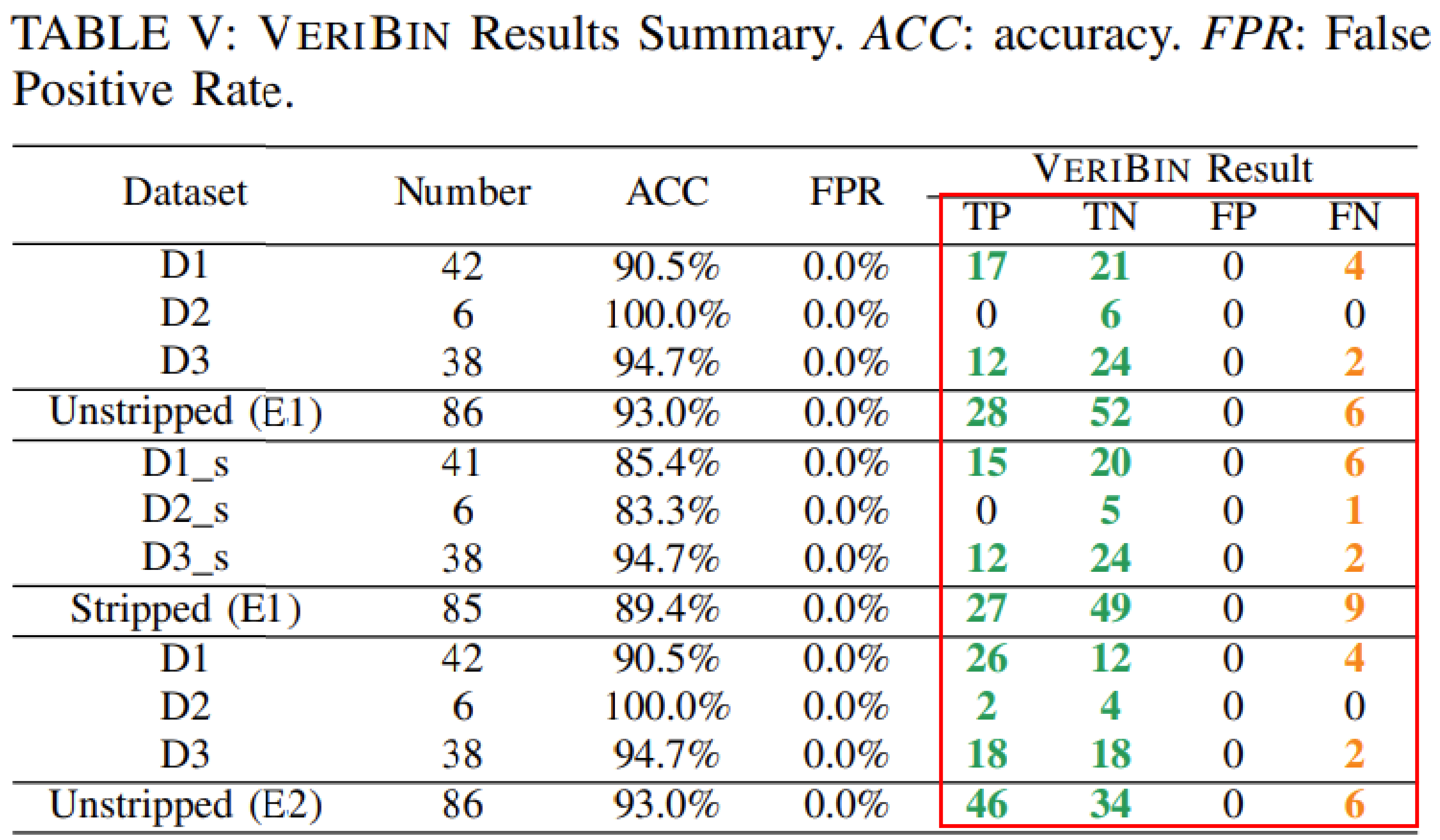
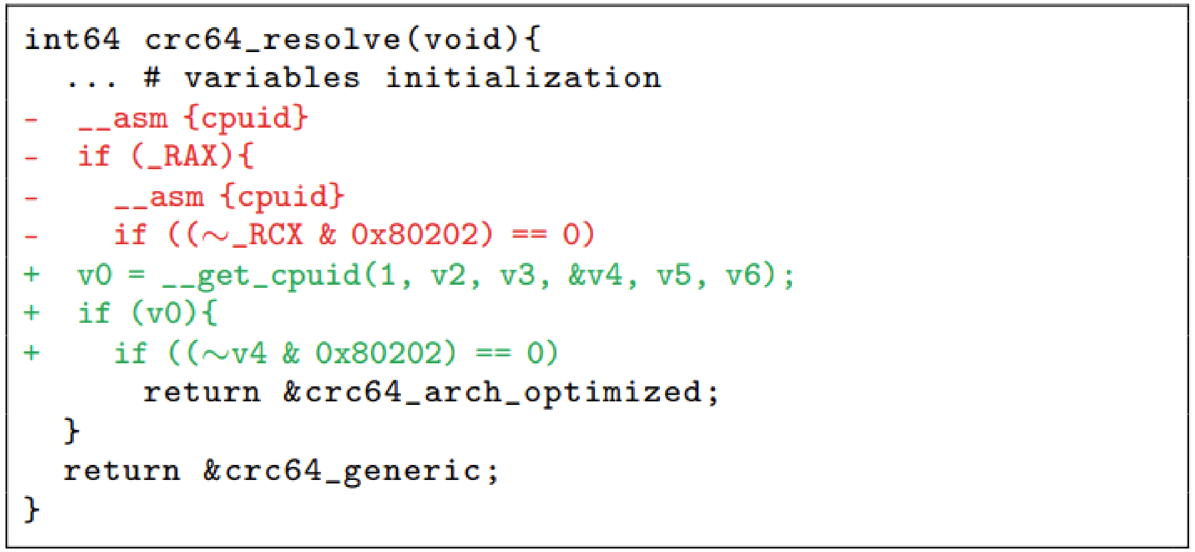


TABLE V: VERIBIN Results Summary. *ACC*: accuracy. *FPR*: False Positive Rate.

Dataset	Number	ACC	FPR	VERIBIN Result			
				TP	TN	FP	FN
D1	42	90.5%	0.0%	17	21	0	4
D2	6	100.0%	0.0%	0	6	0	0
D3	38	94.7%	0.0%	12	24	0	2
Unstripped (E1)	86	93.0%	0.0%	28	52	0	6
D1_s	41	85.4%	0.0%	15	20	0	6
D2_s	6	83.3%	0.0%	0	5	0	1
D3_s	38	94.7%	0.0%	12	24	0	2
Stripped (E1)	85	89.4%	0.0%	27	49	0	9
D1	42	90.5%	0.0%	26	12	0	4
D2	6	100.0%	0.0%	2	4	0	0
D3	38	94.7%	0.0%	18	18	0	2
Unstripped (E2)	86	93.0%	0.0%	46	34	0	6

Case Studies (CVE-2024-3094 XZ Utils)



```
int64 crc64_resolve(void){
  ... # variables initialization
-  __asm {cpuid}
-  if (_RAX){
-    __asm {cpuid}
-    if ((~_RCX & 0x80202) == 0)
+  v0 = __get_cpuid(1, v2, v3, &v4, v5, v6);
+  if (v0){
+    if ((~v4 & 0x80202) == 0)
      return &crc64_arch_optimized;
  }
  return &crc64_generic;
}
```

Listing 3: Simplified pseudocode of the malicious modifications in XZ Utils, created by comparing decompiled code from version 5.5.2beta and version 5.6.0 of liblzma.so. In version 5.6.0, the original `cpuid` instruction is replaced with a call to a compromised `__get_cpuid` function. Note that these modifications are not directly visible in the source code.