VeriBin: Adaptive Verification of Patches at the Binary Level

Hongwei Wu, Jianliang Wu*, Ruoyu Wu, Ayushi Sharma, Aravind Machiry, and Antonio Bianchi

Purdue University, Simon Fraser University*

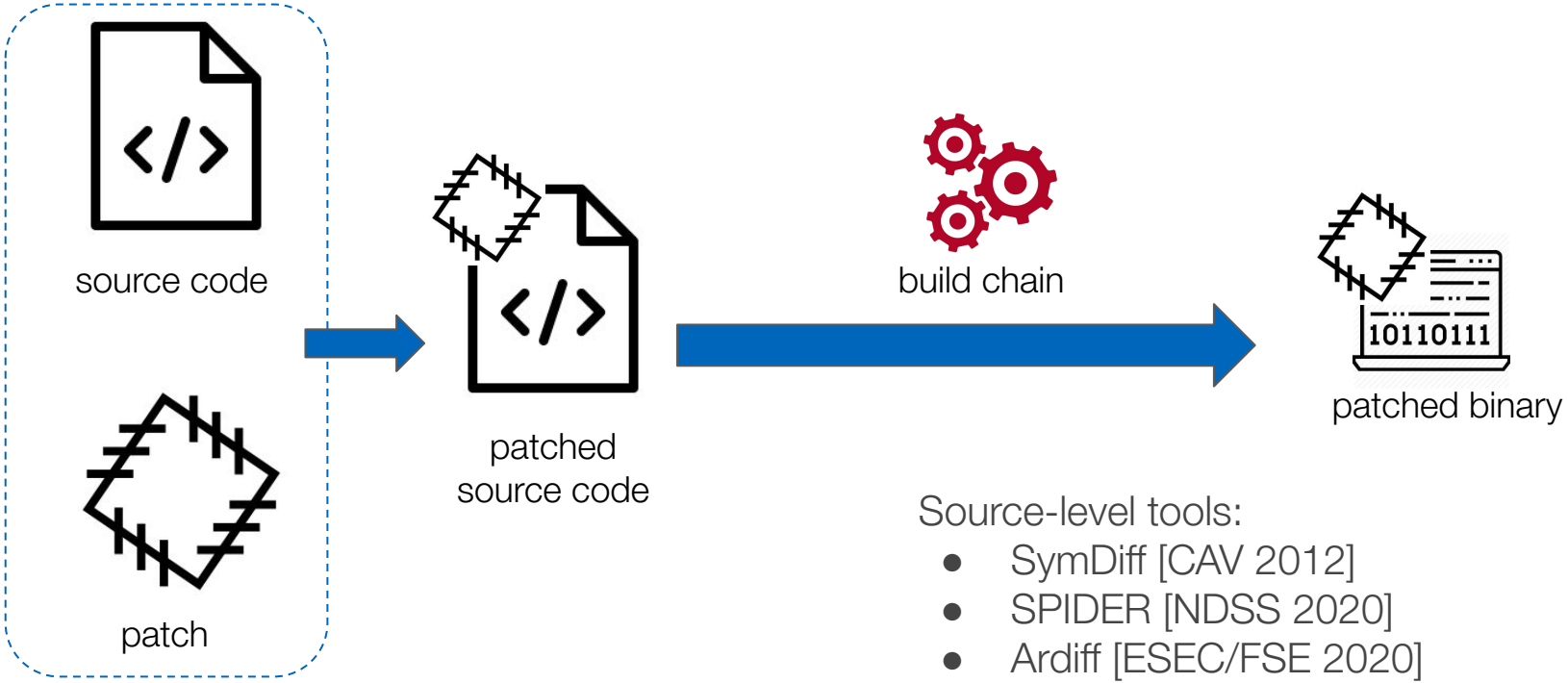
NDSS 2025

Supported by DARPA, as part of the AMP program, under contract number N6600120C4031

Motivation: The Challenge of Patching

- “**If it ain't broke, don't fix it.**” often applies in software.
- The fear of introducing new issues can lead vendors to leave vulnerabilities unpatched
  - Especially in domains where reliability is the primary concern:
    - Automotive
    - Medical devices
    - …

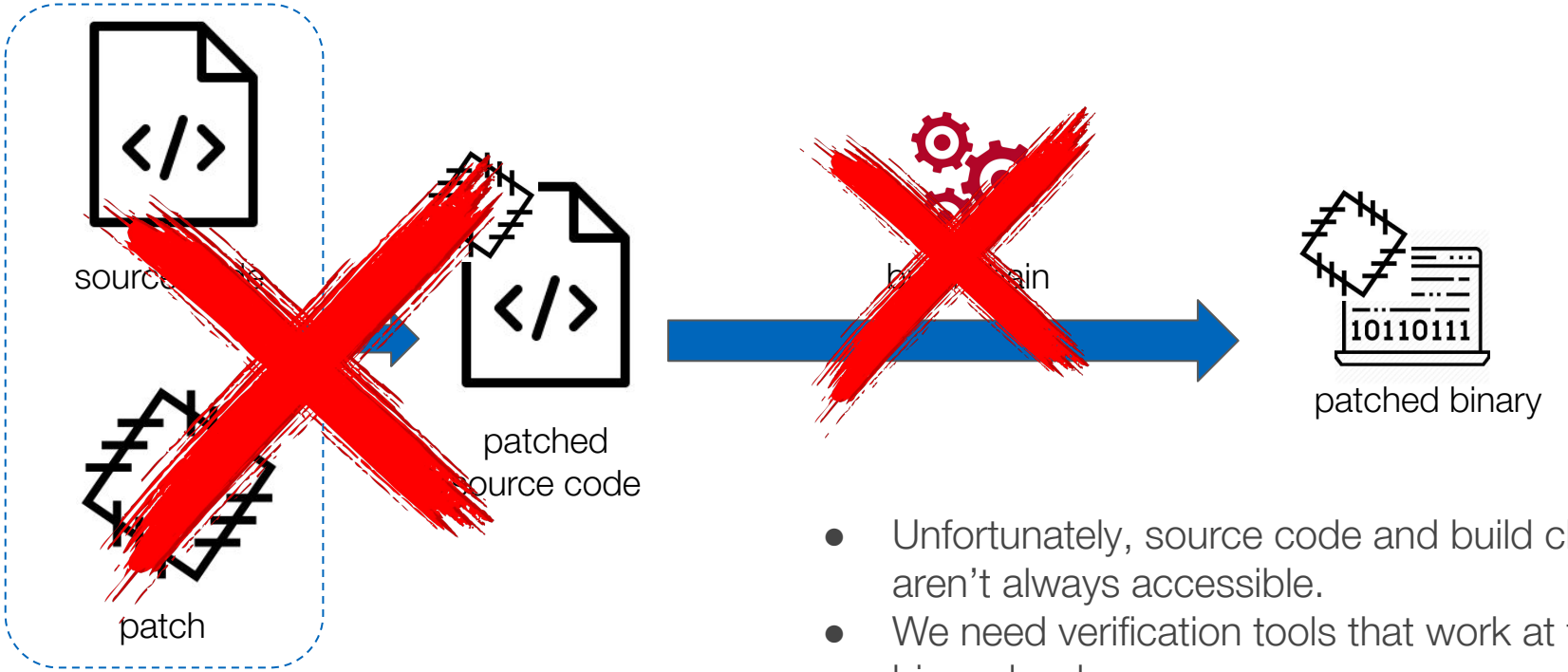
Motivation: source-level patch verification tools



Source-level tools:

- SymDiff [CAV 2012]
- SPIDER [NDSS 2020]
- Ardiff [ESEC/FSE 2020]
- …

Motivation: source-level patch verification tools



- Unfortunately, source code and build chain aren't always accessible.
- We need verification tools that work at the binary level.

Motivation: Why Verify Binary Patches?

- Imagine you receive a patched binary from a third-party vendor. How can you formally guarantee this patched binary:
  - Doesn't break existing functionality?
  - Doesn't introduce unwanted changes or vulnerabilities?

- We need patch verification tools that:
  - **Formally model patch effects** without source code access.
  - Ensure patches **preserve the original binary's functionality**

Traditional ways to verify patches without source

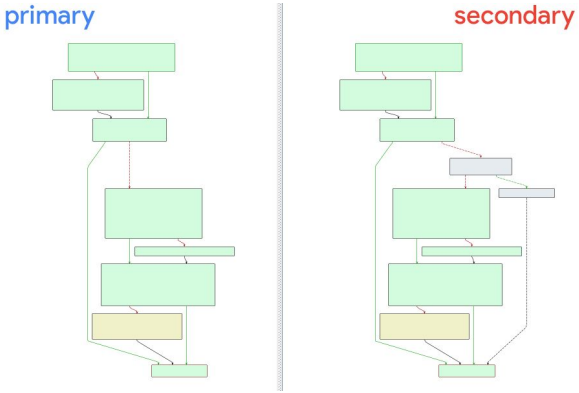
```
original/tidy
0000 0000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF.... ........
0000 0010: 02 00 3E 00 01 00 00 00  00 11 40 00 00 00 00 00  ..>..... ..@.....
0000 0020: 40 00 00 00 00 00 00 00  78 82 0B 00 00 00 00 00  @....... x.......
0000 0030: 00 00 00 00 40 00 38 00  09 00 40 00 25 00 22 00  ....@.8. ..@.%.".
0000 0040: 06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
patched/tidy
0000 0000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF.... ........
0000 0010: 02 00 3E 00 01 00 00 00  00 11 40 00 00 00 00 00  ..>..... ..@.....
0000 0020: 40 00 00 00 00 00 00 00  80 82 0B 00 00 00 00 00  @....... ........
0000 0030: 00 00 00 00 40 00 38 00  09 00 40 00 25 00 22 00  ....@.8. ..@.%.".
0000 0040: 06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
```

(a) Manual Byte Pattern Comparison

Traditional ways to verify patches without source

```
original/tidy
0000 0000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF.... ........
0000 0010: 02 00 3E 00 01 00 00 00  00 11 40 00 00 00 00 00  ..>..... ..@.....
0000 0020: 40 00 00 00 00 00 00 00  78 82 0B 00 00 00 00 00  @.......  x.......
0000 0030: 00 00 00 00 40 00 38 00  09 00 40 00 25 00 22 00  ....@.8. ..@.%.".
0000 0040: 06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
patched/tidy
0000 0000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF.... ........
0000 0010: 02 00 3E 00 01 00 00 00  00 11 40 00 00 00 00 00  ..>..... ..@.....
0000 0020: 40 00 00 00 00 00 00 00  80 82 0B 00 00 00 00 00  @.......  ........
0000 0030: 00 00 00 00 40 00 38 00  09 00 40 00 25 00 22 00  ....@.8. ..@.%.".
0000 0040: 06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
```

(a) Manual Byte Pattern Comparison

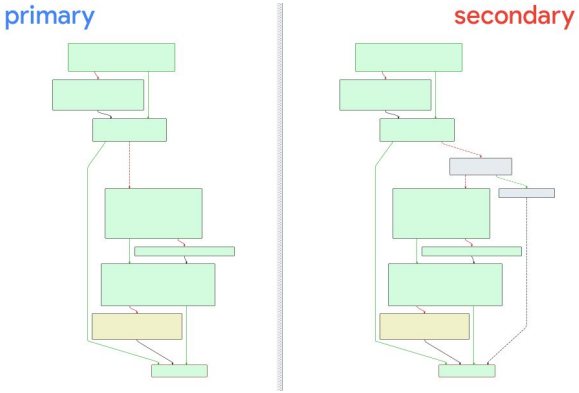


(b) Structural Comparison with BinDiff

Traditional ways to verify patches without source

```
original/tidy
0000 0000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF.... ........
0000 0010: 02 00 3E 00 01 00 00 00  00 11 40 00 00 00 00 00  ..>..... ..@.....
0000 0020: 40 00 00 00 00 00 00 00  78 82 0B 00 00 00 00 00  @....... x.......
0000 0030: 00 00 00 00 40 00 38 00  09 00 40 00 25 00 22 00  ....@.8. ..@.%."
0000 0040: 06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
patched/tidy
0000 0000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF.... ........
0000 0010: 02 00 3E 00 01 00 00 00  00 11 40 00 00 00 00 00  ..>..... ..@.....
0000 0020: 40 00 00 00 00 00 00 00  80 82 0B 00 00 00 00 00  @....... ........
0000 0030: 00 00 00 00 40 00 38 00  09 00 40 00 25 00 22 00  ....@.8. ..@.%."
0000 0040: 06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
```

(a) Manual Byte Pattern Comparison



(b) Structural Comparison with BinDiff

Original

```
__fastcall sub_429F20(__int64 a1)
{
  __int64 result; // rax
  unsigned int v2; // [rsp+14h] [rbp-Ch]
  const char *v3; // [rsp+18h] [rbp-8h]

  if ( *(_QWORD *)(a1 + 4944) )
    sub_427AD2(a1, 0, (__int64)"Doctype given is \"%s\"", *(const char **)(a1 + 4944));
  result = sub_42AD64(a1, 0x15u);
  if ( !(_DWORD)result )
  {
    v2 = sub_418673(a1);
    v3 = (const char *)sub_4186DA(v2);
    if ( !v3 )
      v3 = "HTML Proprietary";
    sub_427AD2(a1, 0, (__int64)"Document content looks like %s", v3);
    result = sub_4186FC(a1);
    if ( (_DWORD)result )
      return sub_427AD2(a1, 0, (__int64)"No system identifier in emitted doctype");
  }
  return result;
}
```

Patched

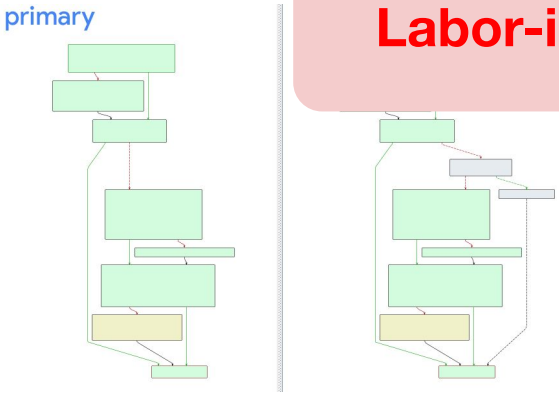
```
__fastcall sub_429F20(__int64 a1)
__int64 result; // rax
unsigned int v2; // [rsp+14h] [rbp-Ch]
const char *v3; // [rsp+18h] [rbp-8h]

if ( *(_QWORD *)(a1 + 4944) )
  sub_427AD2(a1, 0, (__int64)"Doctype given is \"%s\"", *(const char **)(a1 + 4944));
result = sub_42AD78(a1, 0x15u);
if ( !(_DWORD)result )
{
  result = *(_QWORD *)(a1 + 104);
  if ( result )
  {
    v2 = sub_418673(a1);
    v3 = (const char *)sub_4186DA(v2);
    if ( !v3 )
      v3 = "HTML Proprietary";
    sub_427AD2(a1, 0, (__int64)"Document content looks like %s", v3);
    result = sub_4186FC(a1);
    if ( (_DWORD)result )
      return sub_427AD2(a1, 0, (__int64)"No system identifier in emitted doctype");
  }
}
return result;
}
```

(c) Manual Decompiled Code Comparison

Traditional ways to verify patches without source

```
original/tidy
0000 0000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF.... ........
0000 0010: 02 00 3E 00 01 00 00 00  00 11 40 00 00 00 00 00  ..>..... ..@.....
0000 0020: 40 00 00 00 00 00 00 00  78 82 0B 00 00 00 00 00  @....... x.......
0000 0030: 00 00 00 00 40 00 38 00  09 00 40 00 25 00 22 00  ....@.8. ..@.%."
0000 0040: 06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
patched/tidy
0000 0000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF.... ........
0000 0010: 02 00 3E 00 01 00 00 00  00 11 40 00 00 00 00 00  ..>..... ..@.....
0000 0020: 40 00 00 00 00 00 00 00  80 82 0B 00 00 00 00 00  @....... ........
0000 0030: 00 00 00 00 40 00 38 00  09 00 40 00 25 00 22 00  ....@.8. ..@.%."
0000 0040: 06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
```

(a) Manual Byte Pat



(b) Structural Comparison with BinDiff

Labor-intensive and Error-prone

Original

```
__fastcall sub_429F20(__int64 a1)
{
  __int64 result; // rax
  unsigned int v2; // [rsp+14h] [rbp-Ch]
  const char *v3; // [rsp+18h] [rbp-8h]

  if ( *(_QWORD *)(a1 + 4944) )
    sub_427AD2(a1, 0, (__int64)"Doctype given is \"%s\"", *(const char **)(a1 + 4944));
  result = sub_42AD64(a1, 0x15u);
  if ( !(_DWORD)result )
  {
    v2 = sub_418673(a1);
    v3 = (const char *)sub_4186DA(v2);
    if ( !v3 )
      v3 = "HTML Proprietary";
    sub_427AD2(a1, 0, (__int64)"Document content looks like %s", v3);
    result = sub_4186FC(a1);
    if ( (_DWORD)result )
      return sub_427AD2(a1, 0, (__int64)"No system identifier in emitted doctype");
```

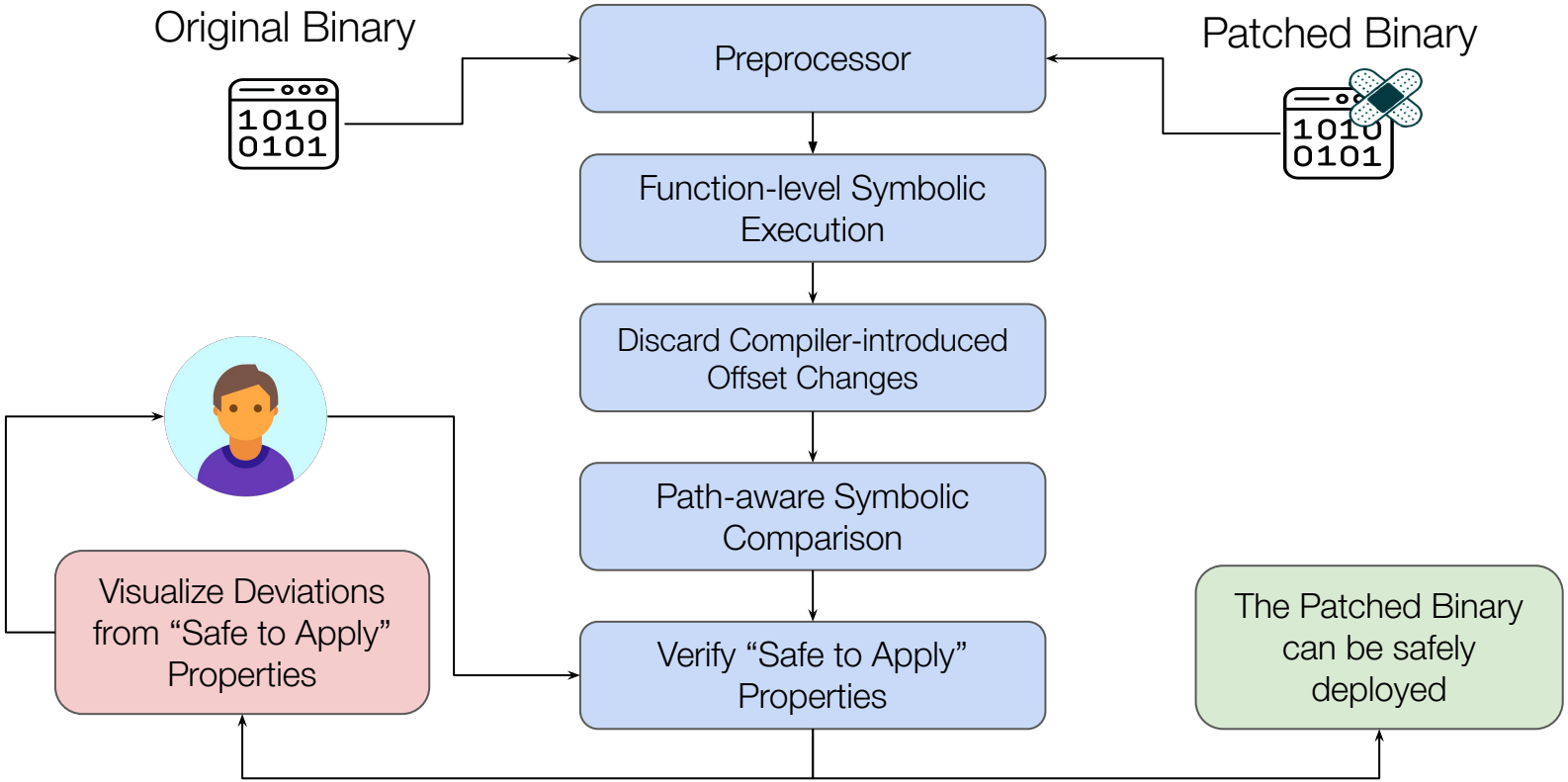
```
    sub_427AD2(a1, 0, (__int64)"Doctype given is \"%s\"", *(const char **)(a1 + 4944));
  result = sub_42AD78(a1, 0x15u);
  if ( !(_DWORD)result )
  {
    result = *(_QWORD *)(a1 + 104);
    if ( result )
    {
      v2 = sub_418673(a1);
      v3 = (const char *)sub_4186DA(v2);
      if ( !v3 )
        v3 = "HTML Proprietary";
      sub_427AD2(a1, 0, (__int64)"Document content looks like %s", v3);
      result = sub_4186FC(a1);
      if ( (_DWORD)result )
        return sub_427AD2(a1, 0, (__int64)"No system identifier in emitted doctype");
    }
  }
  return result;
}
```

(c) Manual Decompiled Code Comparison

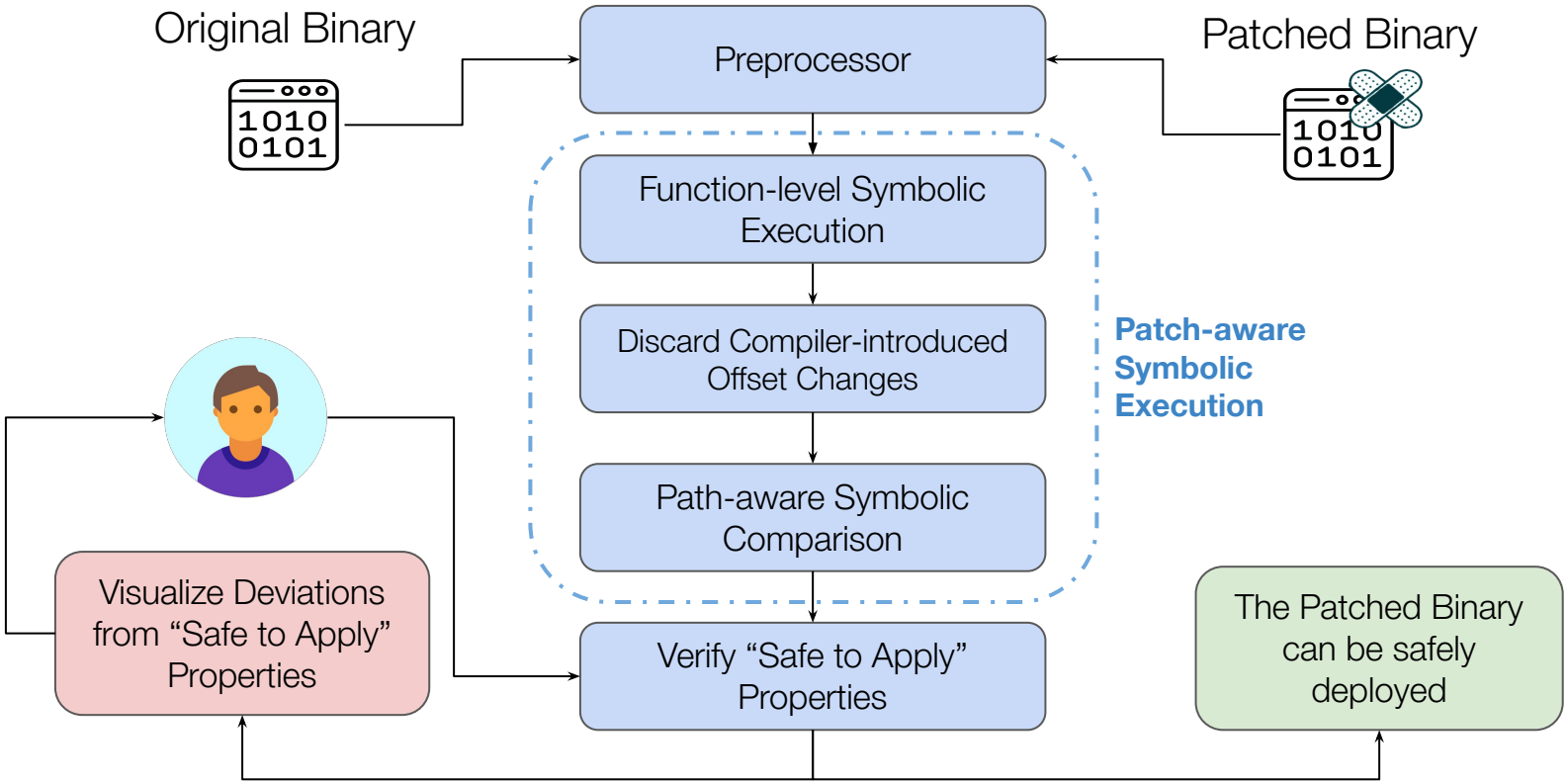
Our tool: VeriBin

- Adaptive Verification of Patches at the Binary Level
  - First system to describe and verify patch behavior at the **binary level** (i.e., without source code), for **functionality-preserving properties**.
  - **Adaptive**: Help analysts to enhance the analysis with domain-specific insights.
  - Addresses unique challenges in binary-level analysis compared to source-level approaches.

Design


Original Binary
1010
0101
Preprocessor
Patched Binary
1010
0101
Function-level Symbolic Execution
Discard Compiler-introduced Offset Changes
Path-aware Symbolic Comparison
Verify “Safe to Apply” Properties
Visualize Deviations from “Safe to Apply” Properties
The Patched Binary can be safely deployed

Design


Original Binary
1010
0101
Preprocessor
Patched Binary
1010
0101
Function-level Symbolic Execution
Discard Compiler-introduced Offset Changes
Path-aware Symbolic Comparison
Patch-aware Symbolic Execution
Visualize Deviations from “Safe to Apply” Properties
Verify “Safe to Apply” Properties
The Patched Binary can be safely deployed

Patch-aware Symbolic Execution

- Using SMT solvers to compare functions faces challenges:
  - **Compiler-Introduced Offset Changes (CIOCs)**: Offsets changes introduced by compilers hinder the verification.
  - **Complicated symbolic expressions are hard to compare:** Type information loss leads to inefficient theories; absence of variable names complicates symbolic value comparisons.

Patch-aware Symbolic Execution

- Using SMT solvers to compare functions faces challenges:
  - **Compiler-Introduced Offset Changes (CIOCs)**: Offsets changes introduced by compilers hinder the verification.
  - **Complicated symbolic expressions are hard to compare:** Type information loss leads to inefficient theories; absence of variable names complicates symbolic value comparisons.

- **VeriBin's solutions :**
  1. **Detect and ignore CIOCs** with the combination methods of *Content-Based Comparison*, *Shift-by-same-offset Analysis* and *Structural Position Correlation*.
  2. Simplify Comparisons: Use ***Matching Path Pairs (MPPs)*** to simplify symbolic comparisons "path by path".

Detect and Discard CIOCs

Compiler-Introduced Offset Change (CIOC): A variation in memory addresses between the original and patched binaries caused by compiler optimizations, despite the content remaining identical.

Why discard CIOCs? They do not reflect actual modifications introduced by the patch.

Source Level Patch

```
char a = ...;
+ char MAX_VAL = ...;
+ if (a > (MAX_VAL - 1){
+     return -1;
+ }
if (checkval(a) != 0) {
    return -1;
}
...
```

Original Binary

```
movsx eax, [rbp - 0x1]
mov edi, eax
call 0x1149
```

Patched Binary

```
movsx edx, [rbp - 0x5]
mov eax, [rbp - 0x4]
cmp edx, eax
jbe ...
...
movsx eax, [rbp - 0x5]
mov edi, eax
call 0x1149
```

Offset changed for variable **a**

Potential Function Call Argument Difference:

- Original binary:
  `call func_1149(mem_[rbp - 0x1])`
- Patched binary:
  `call func_1149(mem_[rbp - 0x5])`

Detect and Discard CIOCs

Techniques to detect CIOCs:

1. **Content-Based Comparison**: Compare contents at fixed addresses for global read-only variables.
   - `printf(0x4000)` v.s. `printf(0x4040)`,
     0x4000 in original binary and 0x4040 in the patched binary both point to the content "abc"
   - `printf(“abc”)` v.s. `printf(“abc”)`

Detect and Discard CIOCs

Techniques to detect CIOCs:

2. **Shift-by-same-offset Analysis**: Identify local variables shifted by a consistent offset.
   - `foo(rsp + 0xfffe, rsp + 0xfff8, …)`

     v.s.

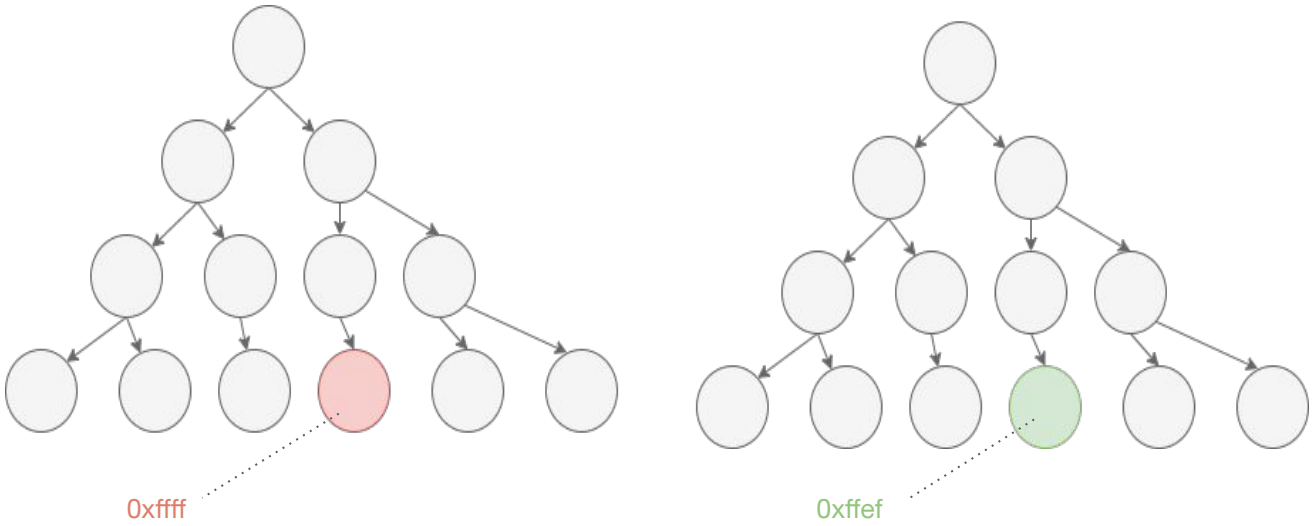
     `foo(rsp + 0xffde, rsp + 0xffd8, …)`

     All variables are shift by the same offset 0x20.

Detect and Discard CIOCs

Techniques to detect CIOCs:

3. **Structural Position Correlation:** Match expressions in similar AST positions differing only by an offset.

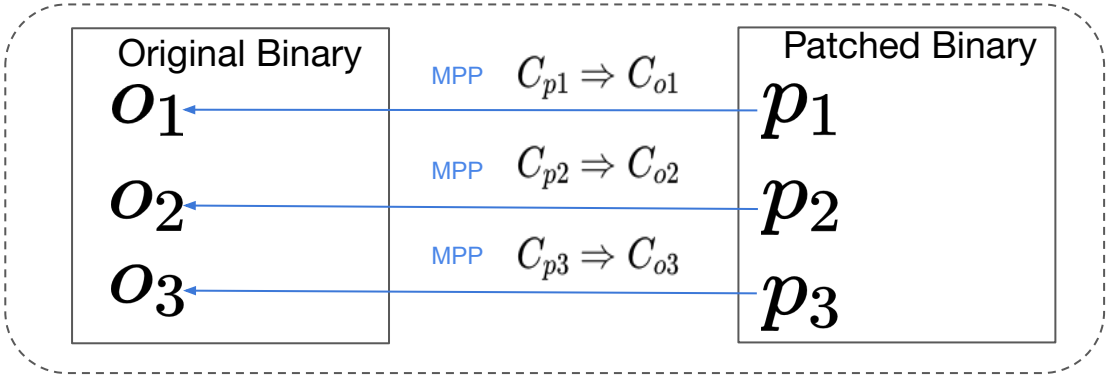


Simplify Comparison by Matching Path Pair (MPP)

Definition: An MPP is a pair of valid exit paths, *o* and *p*, where any input *i* executing *p* in patched function also executes *o* in original function.

In other words, the path constraint of *p* imply that of *o*.

$C_p \Rightarrow C_o$, i.e., $C_p \wedge \neg C_o$ is Unsat.



Simplify Comparison by Matching Path Pair (MPP)

Purpose: MPPs help by avoiding complex symbolic comparisons.

```
switch (choice) {
    case 1:
        ptr->value = 10;
        break;
    case 2:
        ptr->value = 20;
        break;
    case 3:
-       ptr->value = 30;
+       ptr->value = 35;
        break;
    case 4:
        ptr->value = 40;
        break;
    default:
        ptr->value = 0;
        break;
}
```

Comparing `ptr->value`

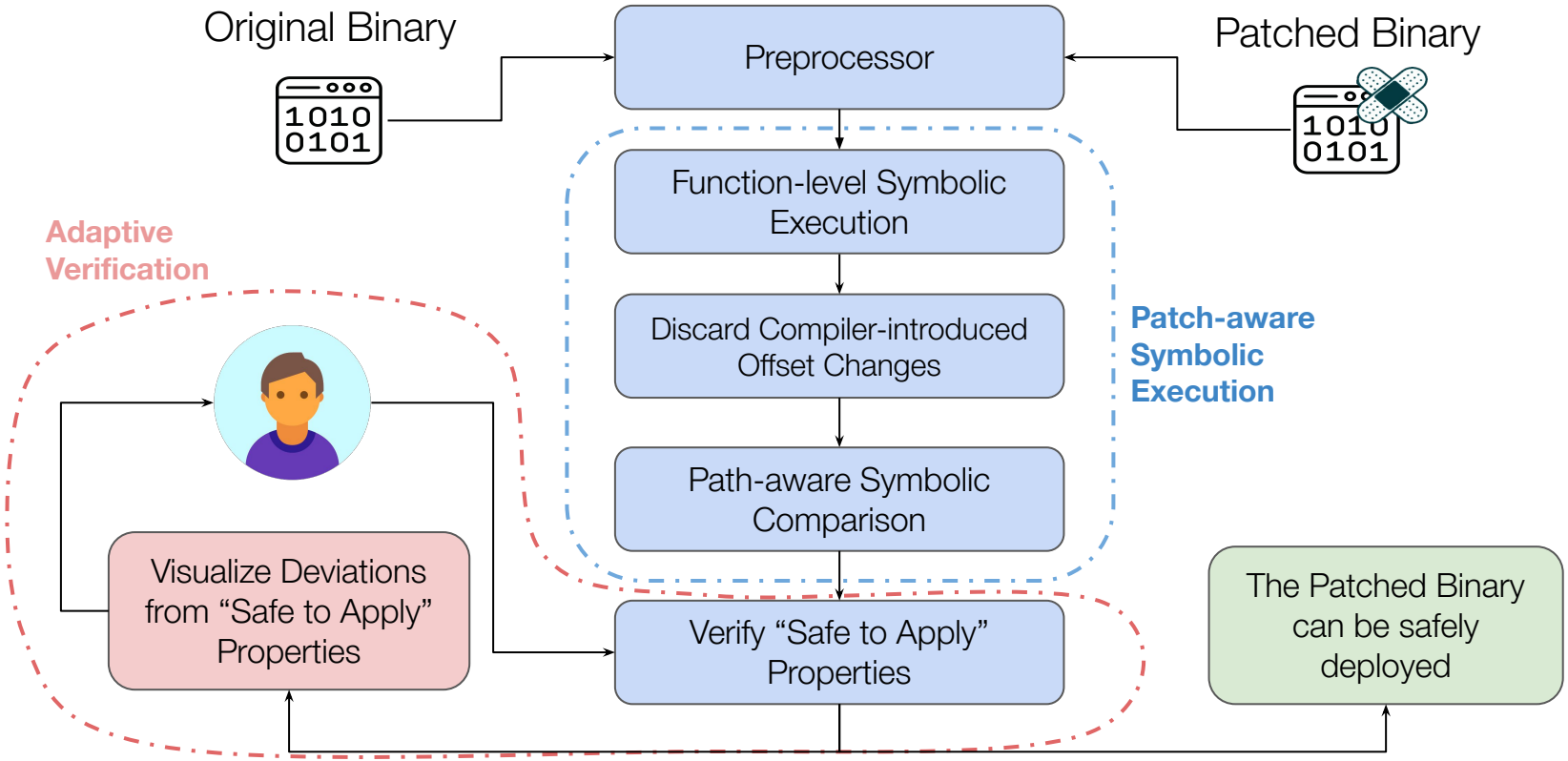
(a) If merging all the paths, SMT solver needs to compare:

```
ptr->value = ITE(choice == 1, 10,
                 ITE(choice == 2, 20,
                     ITE(choice == 3, 30, 0
                         ITE(choice == 4, 40, 0))))
V.S.
ptr->value = ITE(choice == 1, 10,
                 ITE(choice == 2, 20,
                     ITE(choice == 3, 35, 0
                         ITE(choice == 4, 40, 0))))
```

(b) If comparing MPP, SMT solver only compares:

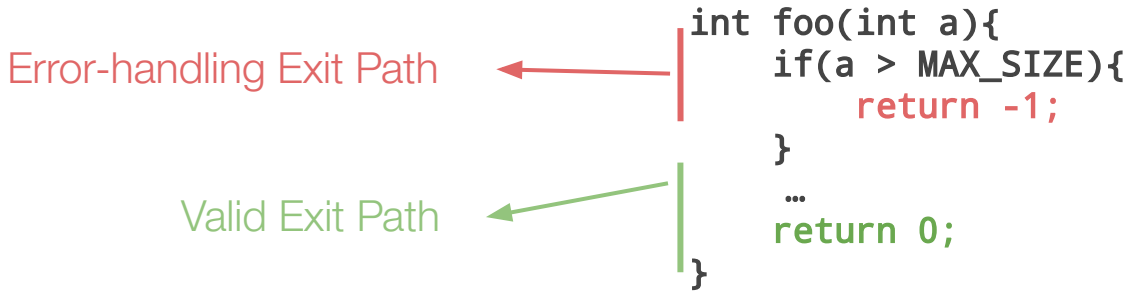
```
ptr->value = 30
V.S.
ptr->value = 35
```

Design: Adaptive Verification



Terminology: Valid Exit Path to a function

- All reachable complete execution paths = VEPs ∪ EEPs
- **Valid Exit Path (VEP)**: A complete path that the function execution takes only with valid inputs.
- **Error-handling Exit Path (EEP)**: A complete path where inputs that follow this path are considered invalid and thus rejected by the function.



Terminology: Safe-to-Apply Properties

To ensure the patch is not breaking the original functionality, VeriBin verifies the following properties for each modified function:

- **Not increasing input space:** All valid inputs to the patched function are also valid inputs to the original function.
  - (P1) Path Constraint Implication (for all valid exit paths)
- **Output Equivalence**: For all valid inputs, the output of the patched function must be the same as that of the original function.
  - (P2) Non-Local Memory Writes Equivalence (for all valid exit paths)
  - (P3) Return Value Equivalence (for all valid exit paths)
  - (P4) Function Calls Equivalence (for all valid exit paths)

[1] "SPIDER: Enabling Fast Patch Propagation in Related Software Repositories", Machiry et al., S&P 2020

Safe-to-Apply Properties Verification

- **VeriBin** automates the verification of Safe-to-Apply (StA) properties:
  - If all properties are **True**, the patch is deemed **safe to apply.**
  - If any StA property fails:
    - **Root causes** of the failures are identified.
    - Analysts are engaged to validate the root cause, and their feedback is used to refine the analysis process to filter out **semantically equivalent changes**.

Adaptive Verification: Example

- Patch substituting the usage of the (weak) **3DES** encryption algorithm with the safe **AES** algorithm.

```
int encrypt(...){
  EVP_CIPHER_CTX *ctx; ...
- if(1!=EVP_EncryptInit_ex(ctx,EVP_des_ede3_cbc(),NULL,key,iv))
+ if(1!=EVP_EncryptInit_ex(ctx,EVP_aes_256_cbc(),NULL,key,iv))
  { handleErrors(); ... }
  ...
}
```

- VeriBin detects the **potential violation** of the Safe to Apply properties and **asks the analyst**:
  - *"Can EVP_des_ede3_cbc() and EVP_aes_256_cbc() be considered equivalent?"*
- If the operator answers: "yes"
  - This information is integrated in the analysis
    → VeriBin determines this patch is Safe to Apply

Evaluation:

- Dataset: 86 pair of original and patched binaries from
  - MicroPatch Bench dataset [1]
  - DARPA AMP Challenges dataset
  - PatchDB dataset [2]
- Evaluated VeriBin on unstripped and stripped versions
  - Unstripped: 93% accuracy, no false positives
  - Stripped: 89.4% accuracy, no false positives
- Average runtime: ~1,300s
  - ~570s for symbolic execution
  - ~640s for verification of StA properties
  - ~90s for other steps

[1] MicroPatch Bench: https://github.com/Aarno-Labs/micropatch-bench
[2] PatchDB: https://sunlab-gmu.github.io/PatchDB/

Case Study: Tidy

- A minimal patch was applied to the *Tidy* binary, adding a check to ensure ***doc->lexer*** is ***not 0***.

```
     if ( ! cfgBool(doc, TidyXmlTags) )
     {
+        if (doc->lexer == 0)
+            return;
         Bool isXhtml = doc->lexer->isvoyager;
         uint apparentVers;
         ctmbstr vers;
```

- **Why it is safe-to-apply:**
  - The patch **restricts** input values by validating *doc->lexer.*
  - No functionality-breaking modifications or side effects are introduced.

- **VeriBin Analysis Results**:
  - All safe-to-apply properties are verified as True → Patch classified as **Safe to Apply**.

Case Study: XZ Backdoor

- The “XZ backdoor” was maliciously introduced in XZ Utils by modifying the build process of the liblzma library

- **VeriBin can easily detect the backdoor**:
  - Assembly instruction **cpuid** is replaced by malicious **__get_cpuid()** function
  - The patch is non-StA

```
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;
}
```

- **Binary-level verification is helpful even if source code is available**

Summary

- VeriBin, a binary-level patch verification tool
  - compare a binary with its patched version
  - determine whether the patch is “Safe to Apply”.
- VeriBin is accurate
  - Unstripped binary: 93% accuracy, no false positives
  - Stripped binary: 89.4% accuracy, no false positives
- VeriBin is open-source: https://github.com/purseclab/VeriBin

Thank you! Questions?

wu1685@purdue.edu