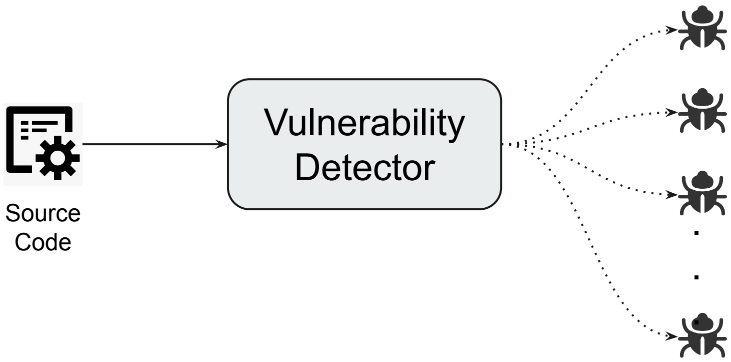
Patch Propagation

Holistic Software Security

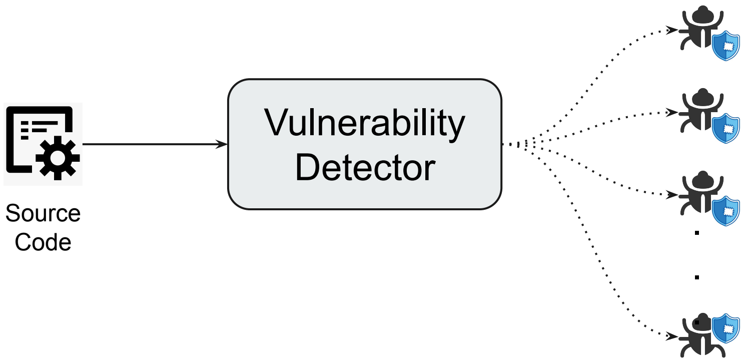
Aravind Machiry

Importance of Patch Propagation



Okay, we found vulnerabilities. Now what?

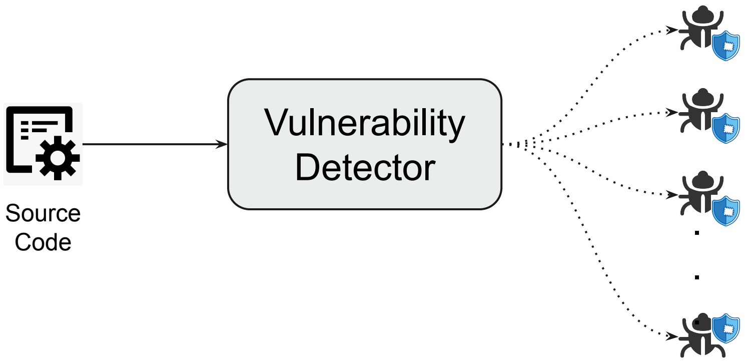
Importance of Patch Propagation



Okay, we found vulnerabilities. Now what?

These vulnerabilities need to be **patched**.

Importance of Patch Propagation

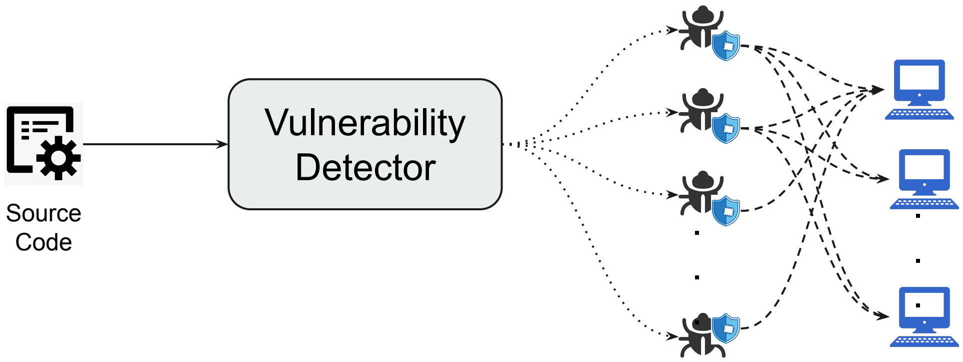


Okay, we found vulnerabilities. Now what?

These vulnerabilities need to be patched.

Is this enough?

Importance of Patch Propagation

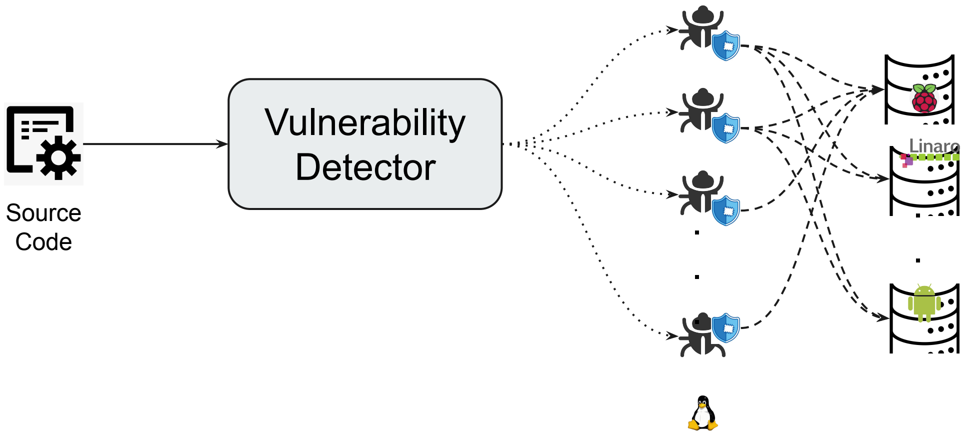


Okay, we found vulnerabilities. Now what?

These vulnerabilities need to be patched.

Patched software need to be pushed to machines.

Importance of Patch Propagation



Okay, we found vulnerabilities. Now what?

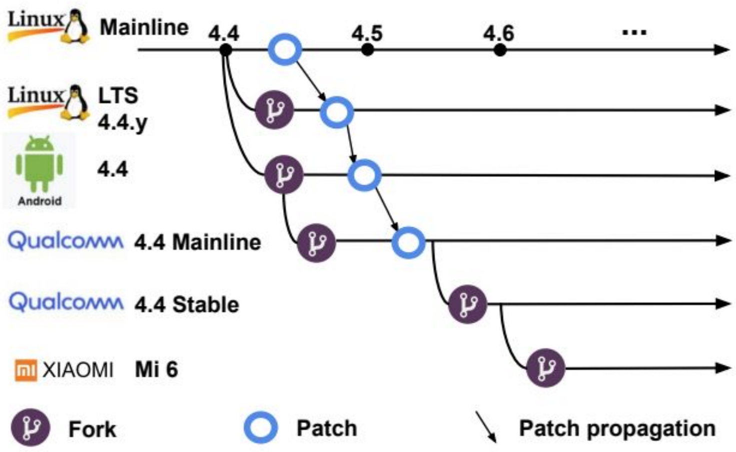
These vulnerabilities need to be patched.

Patched software need to be pushed to machines.

Patches need to be pushed to related repositories.

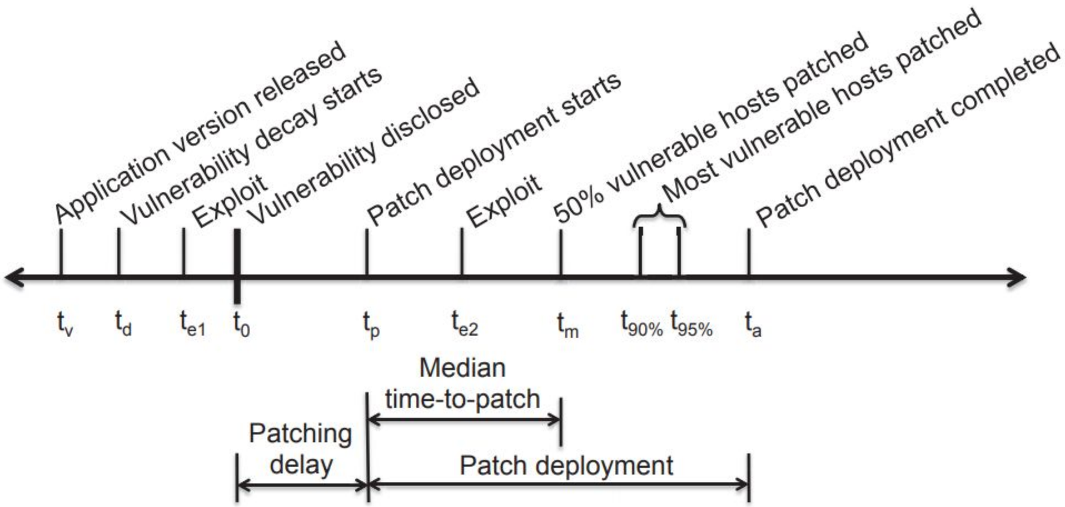
Importance of Patch Propagation

- Software Diversity: Different versions of same software.
- Code clones: Same code used in different platforms.
    - E.g., Linux code in Android, Mac OS code in iOS, etc.



* An Investigation of the Android Kernel Patch Ecosystem

Delays in Patching



* The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching

Delays in Patching

Different vendors have different practices and priorities.

Delay varies across different vendors.

Patch delay [days]		Vendor	Missed Patches 2018	Missed Patches 2019	Samples*
Immediately	0	Google	0 to 0.2	0 to 0.2	many
	0	Sony	0.2 to 1	0.2 to 1	lots
	0	Nokia	0.2 to 1	0.2 to 1	lots
Within 2 weeks	6	Huawei	0.2 to 1	0.2 to 1	lots
	12	LGE	0 to 0.2	0 to 0.2	lots
	14	Samsung	0 to 0.2	0 to 0.2	lots
Within 1 month	15	Motorola	0 to 0.2	0.2 to 1	lots
	15	BQ	0.2 to 1	0.2 to 1	many
	15	ZTE	2 to 4	0 to 0.2	lots
	16	Oppo	4 or more	1 to 2	few
	18	Wiko	2 to 4	0 to 0.2	few
	18	Verizon	0.2 to 1	0 to 0.2	few
	21	Lenovo	4 or more	0 to 0.2	few
	21	TCL	2 to 4	0.2 to 1	few
	23	Asus	0.2 to 1	0.2 to 1	many
	25	OnePlus	0 to 0.2	0.2 to 1	many
	26	Vivo	1 to 2	0.2 to 1	lots
	30	htc	1 to 2	1 to 2	many
	31	Xiaomi	0.2 to 1	0 to 0.2	many

* https://www.srlabs.de/bites/android-patch-gap-2020

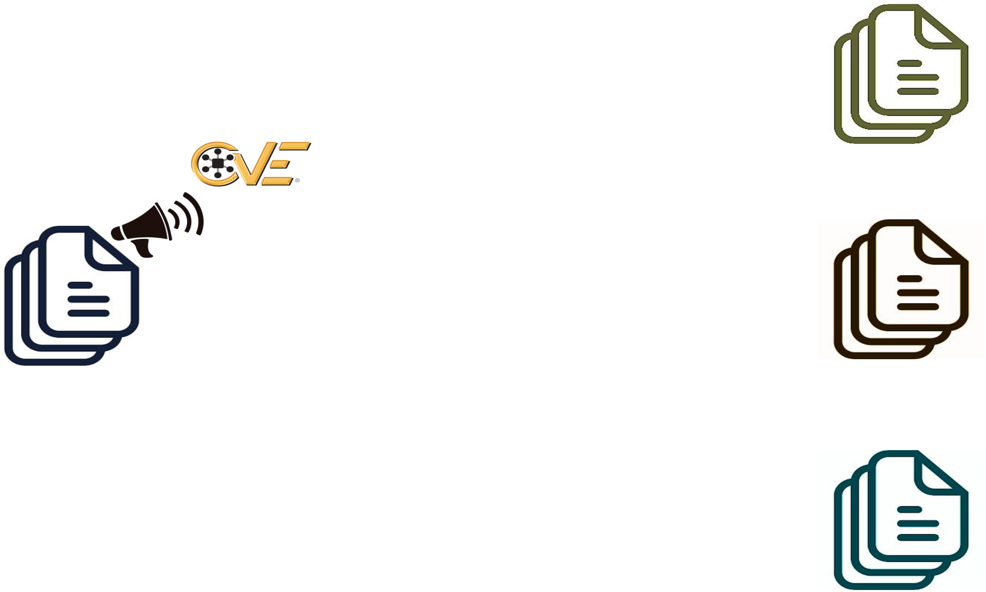
Security Patch Propagation

- Propagation of **security patches should be done ASAP**:
    - To prevent attacker from exploiting it.
    - Ensure that products are secure.
    - To avoid negative publicity.

- How to manage propagation of security patches?

Common Vulnerabilities and Exposures

Security Patch Propagation

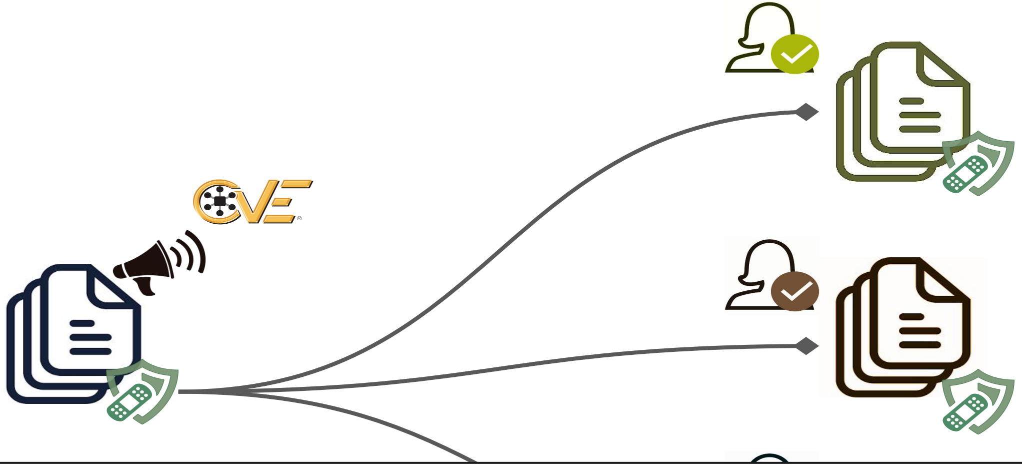


Security Patch Propagation

Security Patch Propagation

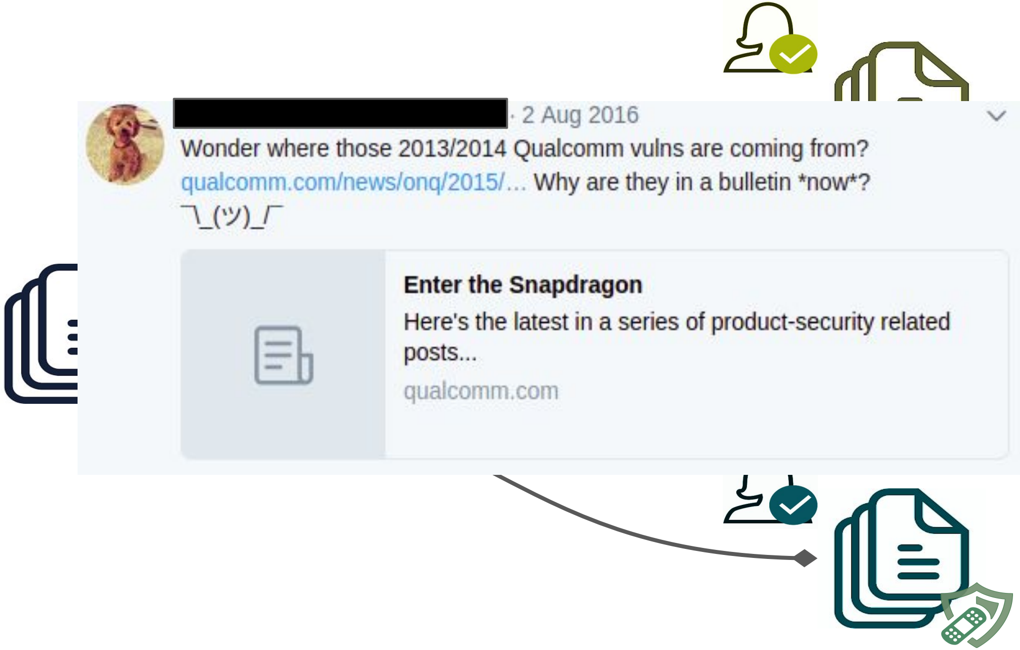
Security Patch Propagation

Security Patch Propagation

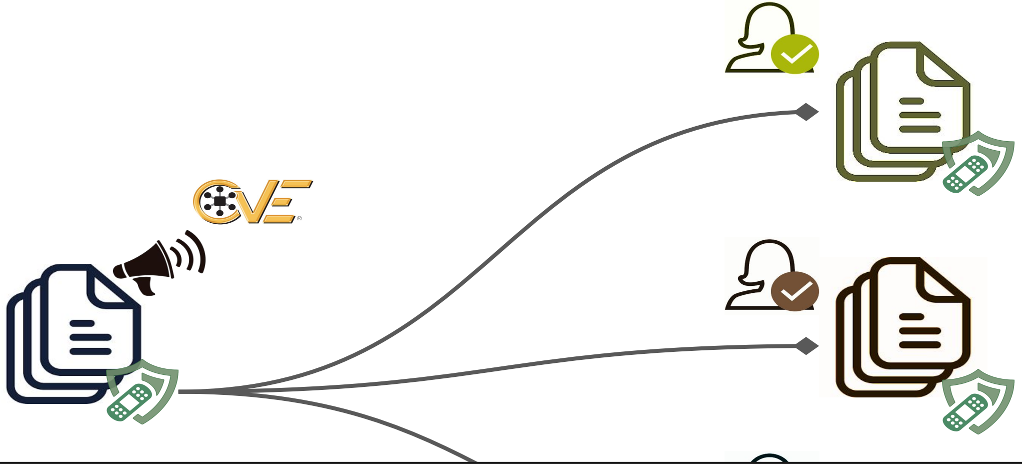


Problem 1: There could be delay in applying patches.
(E.g., Testing after applying patches)

Security Patch Propagation



Security Patch Propagation



Problem 2: Security Patches may not have an assigned CVE number.

Security Patch Propagation

[redacted]

Replying to @videolan and @MITREcorp

So libEBML fixed a vulnerability in 1.3.6, but didn't assign a CVE to it? And as a result, a fully-patched Ubuntu 18.04 system provides a vulnerable 1.3.5 version?

2:16 PM · Jul 24, 2019 · Twitter Web App

Prob ned
CVE

Security Patch Propagation

[redacted]

Why there are at least 6,000 vulnerabilities without CVE-IDs

Posted by Synopsys Editorial Team on Thursday, September 22nd, 2016

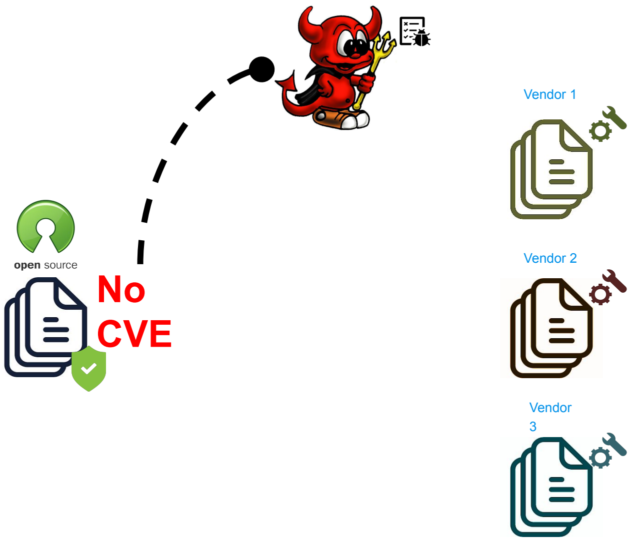
Prob ned

CVE

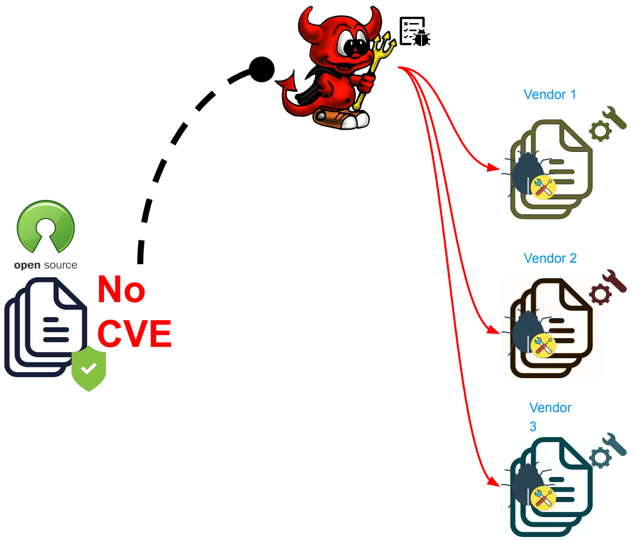
2:16 PM · Jul 24, 2019 · Twitter Web App

Security Patches with no CVE

Security Patches with no CVE



Security Patches with no CVE



How are CVE numbers assigned?

- They need to be requested from CVE Numbering Authorities (CNA):
    - A bit tedious approach.
    - Developers may underestimate the severity of a bug.
    - OSS : Developers raising a pull request might not care about CVEs.

- Distributed Weakness Filing (DWF): New system for vulnerable IDs.

How can we handle this?

- What is the problem?
  - Identification of security patches is done manually by assigning CVE numbers.
  - **Can we identify security patches without CVE numbers?**

Identifying Security Patches automatically

- **Systematic approaches:**
  - Analyze the patch to determine the changes done by the patch => If changes are security related then => Okay.
    - SPIDER => Based on syntactic analysis.
    - SID => Based on semantics.

- **Pattern based or ML approaches:**
  - Given a patch say that it is a security patch.

SPIDER - Intuition

“Verification technique to automatically identify patches (safe patches) that do not adversely affect the functionality of the program”.

Assumption: Most of the security patches are point fixes and do not hugely affect the program functionality.

Safe Patch Should Not Affect the Functionality

- For all **expected inputs**:
    - The output of the patched program should be the same as that of original program.

Safe Patch Should Not Affect the Functionality

- For all **expected inputs**:
  - The output of the patched program should be the same as that of original program.

```
switch (input) {
        ...
        case ...
        case …
 +      case NEW_INPUT: do_something(); break;
        case …
        ...
}
```

Safe Patch Should Not Affect the Functionality

- For all **expected inputs**:
  - The output of the patched program should be the same as that of original program.

```
switch (input) {
```

This patch might break the program on “NEW_INPUT”

```
        case …
 +      case NEW_INPUT: do_something(); break;
        case …
        ...
}
```

Safe Patch Should Not Affect the Functionality

- For all **expected inputs**:
    - The output of the patched program should be the same as that of original program.
- The patch should not allow new inputs into the program.

Safe Patch Should Not Affect the Functionality

- For all **expected inputs**:
  - The output of the patched program should be the same as that of original program.
- The patch should not allow new inputs into the program.

```
if (a >= MAX_LEN) return -1;
```

Safe Patch Should Not Affect the Functionality

- For all **expected inputs**:
  - The output of the patched program should be the same as that of original program.
- The patch should not allow new inputs into the program.

This is **OKAY**. We are restricting inputs (i.e., not allowing new inputs)

```
if (a >= MAX_LEN) return -1;
```

Safe Patches Conditions

A Safe Patch should have:

- **Non-increasing input space (C1):** The patch *should not increase the valid input* space of the program.
- **Output equivalence (C2):** For all the valid inputs, *the output of the patched program must be the same as that of the original program.*

Safe Patches at Function Level

For all functions affected by the patch:

if C1 and C2 holds ⇒ C1 and C2 hold for the entire program.

Non-Increasing Input Space (C1)

The patch should not increase the valid input space of a function.

In other words, All *valid inputs to the patched function (Fp) should also be valid inputs to the original function (Fo)*.

for all inputs *i* : *valid_input(i, Fp)* → *valid_input(i, Fo)*

Valid Inputs to a Function

- Invalid Inputs : Inputs that are treated as invalid by the function i.e., Inputs that reach invalid exit points.

- Valid Inputs : Inputs that reach valid exit points.

Valid Inputs to a Function

- Invalid Inputs : Inputs that are treated as invalid by the function i.e., Inputs that reach invalid exit points.

- Valid Inputs : Inputs that reach valid exit points.

```
int foo(unsigned a) {
    if (a >= MAX_SIZE) {
        return -1;
    }
    ..
    return 0;
}
```

Valid Inputs to a Function

All inputs that can reach valid exit points : **Identify Path Constraints (PC) through Control dependencies.**

```
int foo(unsigned a) {
    if (a >= MAX_SIZE) {
        return -1;
    }
    ..
    return 0;
}
```

Valid Exit Point:
return 0

Inputs that can reach the valid exit point:
PC = !(a >= MAX_SIZE)

Valid Inputs to a Function

$$\text{vinputs}(f) = \bigvee_{i \in \text{VEP}(f)} \text{PC}(i)$$

Valid inputs (*vinputs*) of function (*f*) is the disjunction (*⋁*) of the path constraint (*PC*) of all valid exit points (*VEP*).

Verifying C1 on a Function

Patched function : *Fp*

Original function : *Fo*

vinputs (Fp) ⟶ vinputs (Fo)

Verifying Output Equivalence (C2)

For all the valid inputs, the **output** of the patched function must be the same as that of the original function.

$$\mathbf{i \in vinputs(f_p): output(f_p, i) == output(f_o, i)}$$

Verifying Output Equivalence (C2)

- Output of a function:
  - Return value.
  - Writes to non-local data, i.e., heap and globals.
  - Function calls along with the arguments.
- **Changes in Error handling code does not affect output**

Verifying Output Equivalence (C2)

- Output Depends on the Data flow path:

Data Path 1 (D1):

`(a < 10) is false`

```
int bar(unsigned a) {
    a = baz();
    if (a < 10) {
        a = b + 9;
    }
    ..
    return a;
}
```

Verifying Output Equivalence (C2)

- Output Depends on the Data flow path:

Data Path 2 (D2):

`(a < 10) is true`

```
int bar(unsigned a) {
    a = baz();
    if (a < 10) {
        a = b + 9;
    }
    ..
    return a;
}
```

Verifying Output Equivalence (C2)

$$\forall (Di, Oi) \in output(Fp),$$
$$\exists\ (Dj, Oj) \in output(Fo) \vdash (Oi == Oj) \land (Di \rightarrow Dj)$$

```
int process_req(struct usr_req *req) {
    void *buf;
    size_t msg_sz;
-   if(!req) {
+   if(!req||!req->buff || req->len>MAX_MSG_SIZE) {
        return -EINVAL;
    }
    msg_sz = req->len;
    if(msg_sz % CHUNK_SZ) {
        msg_sz = ((msg_sz/CHUNK_SZ) + 1) * CHUNK_SZ;
    }
    buf = kzalloc(msg_sz + HDR_SIZE, GFP_KERNEL);
    if(buf) {
-       if(!req->buff) {
-           return -EINVAL;
-       }
        if(proc_from_user(buf + HDR_SIZE, req->buff, req->len)) {
+           kfree(buf);
            return -EINVAL;
        }
        kfree(buf);
        return 0;
    }
    return -ENOMEM;
}
```

```
int process_req(struct usr_req *req) {
    void *buf;
    size_t msg_sz;
-   if(!req) {
        return -EINVAL;
    }
    msg_sz = req->len;
    if(msg_sz % CHUNK_SZ) {
        msg_sz = ((msg_sz/CHUNK_SZ) + 1) * CHUNK_SZ;
    }
    buf = kzalloc(msg_sz + HDR_SIZE, GFP_KERNEL);
    if(buf) {
-       if(!req->buff) {
-           return -EINVAL;
-       }
        if(proc_from_user(buf + HDR_SIZE, req->buff, req->len)) {
            return -EINVAL;
        }
        kfree(buf);
        return 0;
    }
    return -ENOMEM;
}
```

Error Exit Points

Valid Exit Point

Valid Inputs to Old Function

!(!(req != 0)) ^

(buf != 0) ^

!(!(req->buff != 0)) ^

!(proc_from_user(buf + HDR_SIZE, req->buff, req->len) != 0)

Valid Inputs to New Function

```
int process_req(struct usr_req *req) {
   void *buf;
   size_t msg_sz;
+  if(!req||!req->buff || req->len>MAX_MSG_SIZE) {
         return -EINVAL;
   }
   msg_sz = req->len;
   if(msg_sz % CHUNK_SZ) {
         msg_sz = ((msg_sz/CHUNK_SZ) + 1) * CHUNK_SZ;
   }
   buf = kzalloc(msg_sz + HDR_SIZE, GFP_KERNEL);
   if(buf) {
         if(proc_from_user(buf + HDR_SIZE, req->buff, req->len)) {
+              kfree(buf);
               return -EINVAL;
          }
         kfree(buf);
         return 0;
   }
   return -ENOMEM;
}
```

Error Exit Points

Valid Exit Point

! (!(req != 0) || !(req->buff != 0) || req->len > MAX_MSG_SIZE) ^

(buf != 0) ^

!(proc_from_user(buf + HDR_SIZE, req->buff, req->len) != 0)

Convert Path Constraint to Symbolic Expression (Old Function)

Use same symbolic variables for unaffected program variables.

S1 S2 S3

Path Constraint (Old function): (!(!(req != 0)) ^ (buf != 0) ^ !(!(req->buff != 0)) ^ !(proc_from_user(buf + HDR_SIZE, req->buff, req->len) != 0))

S4

vinputs (original) = (S1 != 0) && (S2 != 0) ^ (S3 != 0) ^ !(S4 != 0)

Convert Path Constraint to Symbolic Expression (Patched Function)

Use same symbolic variables for unaffected program variables.

S1 S3 S6

S7

Path Constraint (New function): (!(!(req != 0) || !(req->buff == 0) || req->len > MAX_MSG_SIZE) ^ (buf != 0) ^ !(proc_from_user(buf + HDR_SIZE, req->buff, req->len) != 0))

S2 S4

vinputs (patched) = (S1 != 0) ^ (S3 != 0) ^ (S6 <= S7) ^ (S2 != 0) ^ !(S4 != 0)

Verifying Non-Increasing Input Space (C1)

vinputs (patched) ⟶ vinputs (original)

((S1 != 0) ^ (S3 != 0) ^ (S6 <= S7) ^ (S2 != 0) ^ !(S4 != 0)) ⟶ ((S1 != 0) && (S2 != 0) ^ (S3 != 0) ^ !(S4 != 0))

(A ^ B) ⟶ (B)

```
int process_req(struct usr_req *req) {
    void *buf;
    size_t msg_sz;
-   if(!req) {
+   if(!req||!req->buff || req->len>MAX_MSG_SIZE) {
        return -EINVAL;
    }
    msg_sz = req->len;
    if(msg_sz % CHUNK_SZ) {
        msg_sz = ((msg_sz/CHUNK_SZ) + 1) * CHUNK_SZ;
    }
    buf = kzalloc(msg_sz + HDR_SIZE, GFP_KERNEL);
    if(buf) {
-       if(!req->buff) {
-           return -EINVAL;
-       }
        if(proc_from_user(buf + HDR_SIZE,
+           kfree(buf);
            return -EINVAL;
        }
        kfree(buf);
        return 0;
    }
    return -ENOMEM;
}
```

This statement affects output but it is in error-handling block

SID: Another systematic technique

- Based on under-constrained symbolic execution of original and patched program:
    - Determine if patch prevents a security violation which is present in the original program.
    - Based on LLVM => Requires buildable sources.
    - Better guarantees than SPIDER => Deeper reasoning.

Security Patch Identification: Requirements

- R1: In real world, we only have commit i.e., old file and new file:
  - The system should rely on only original and the patched file without additional information (e.g., commit message, build environment, etc).
- R2: We want to identify commits quickly and the system should be easy to deploy:
  - Be fast, lightweight and scalable.
- R3: Similar to vulnerability detection : No false positives, Okay with false negatives:
  - False negatives: Misses identifying security patch => Current state.
  - False positives: Incorrectly marks a patch as security patch => Wrongly propagate the patch.

SPIDER v/s SID

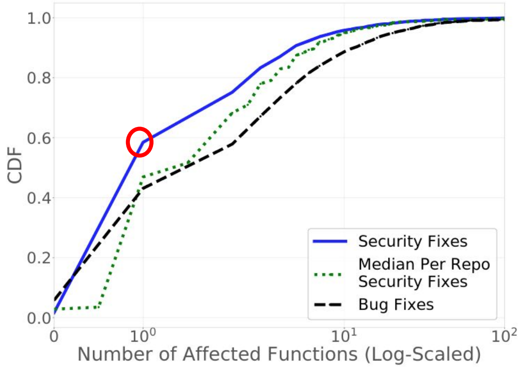
SPIDER	SID
Works only with old file and new file.	Need entire build system => LLVM.
Syntax based: Fast, lightweight and scalable.	Semantic based: UC Symex, relatively slow.
Overly conservative: Misses many patches.	Identifies most of the security patches.
General: Function based => works for all C source files.	Need to perform whole program analysis => Project based.

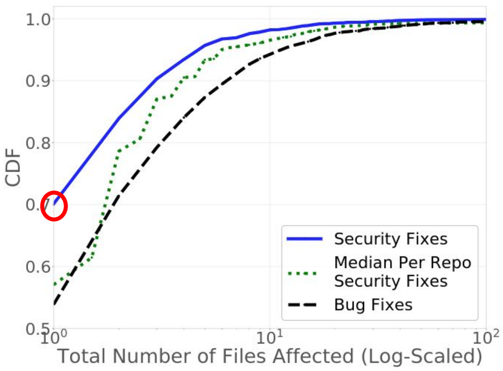
Pattern Based or ML approaches

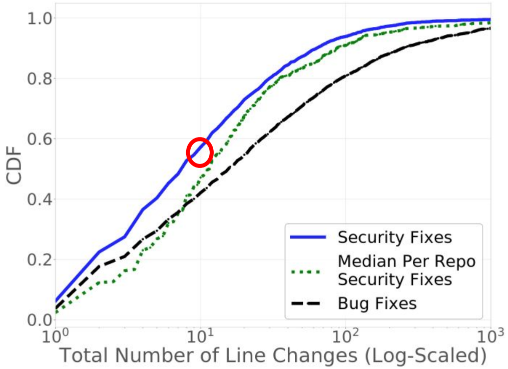
- Intuition: Security patches have distinguishing features.
  - Can we use these features to identify security patches automatically?

Characteristics of security patches!

- Security patches are relatively small!!







* A Large-Scale Empirical Study of Security Patches

Characteristics of security patches!

- Security patches have a specific format!

```
1.+ Security_op(CV, ...)
...
2.  Vulnerable_op(CV, ...)
```

* Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison

ML Based Detection

- Need dataset.
- Feature engineering:
  - Code features
  - Metadata features:
    - Num of files, functions, words in commit message, etc.

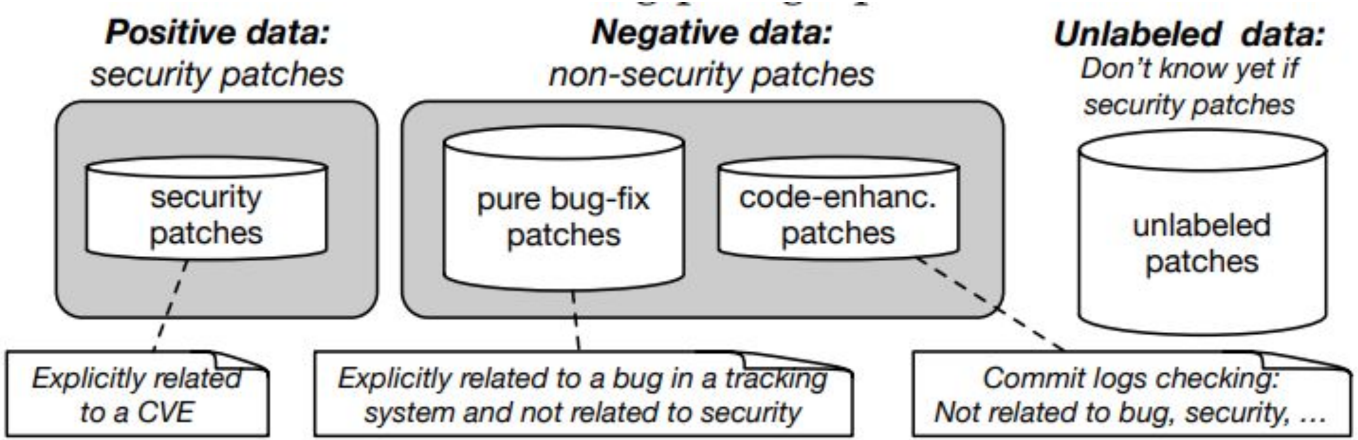
Security Patch Detection by Co-training

- Need dataset => Start from initial dataset , build a model and generate more..repeat.
- Feature engineering:
  - Code features: Num of pointers modified, if/else, loops, sizeof, etc
  - Metadata features: Words in commit message.

* Learning to Catch Security Patches

Security Patch Detection by Co-training

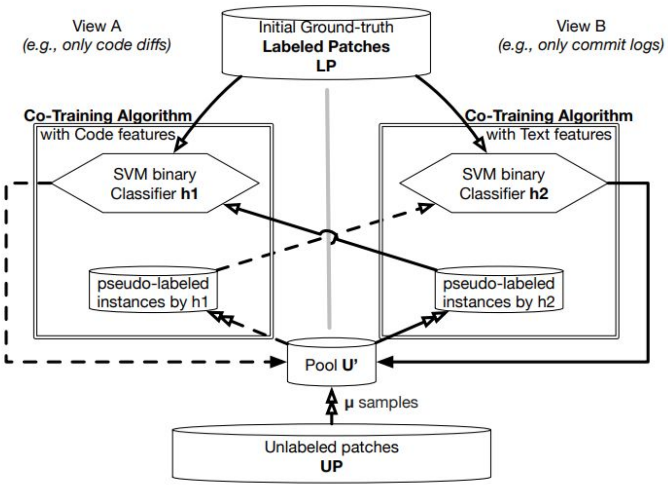
- Initial Dataset



* Learning to Catch Security Patches

Security Patch Detection by Co-training

- Co-training



* Learning to Catch Security Patches

Patch Propagation: Final Remarks

- Very important, yet ignored problem.
- Practicality is very important => Implement your technique as a GitHub Webhook.
- Should have almost no false positives.
- Mailing lists => Unexplored area!