Predicting Security Attacks in FOSS

Why you want it and one way to do it

C.E. Budde R. Paramitha F. Massacci

Università di Trento (IT) & Vrije Universiteit (NL)

Vuln4Cast 2023 FIRST Technical Colloquium

Talk overview

1. Introduction
2. Background
3. Forecast model
4. Conclusions

Introduction

The myth of the bleeding edge

Why You Should Update All Your Software

Updates may sometimes be painful, but they're necessary to keep your devices and data secure on a dangerous internet.

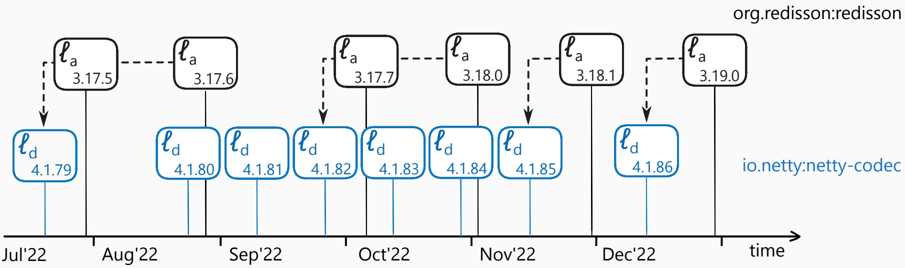
BY **CHRIS HOFFMAN** PUBLISHED AUG 28, 2020

Quick Links

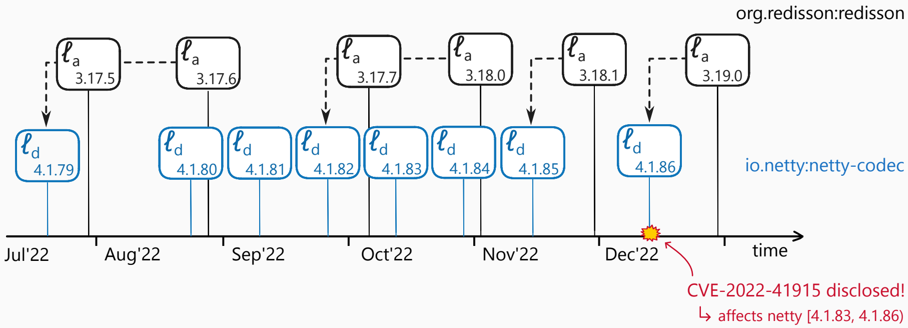
Security Updates 101

What's the Risk, Really?

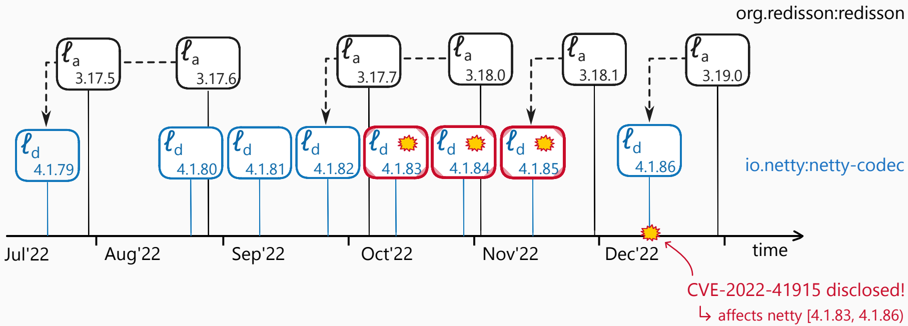
Some motivation (plz!)



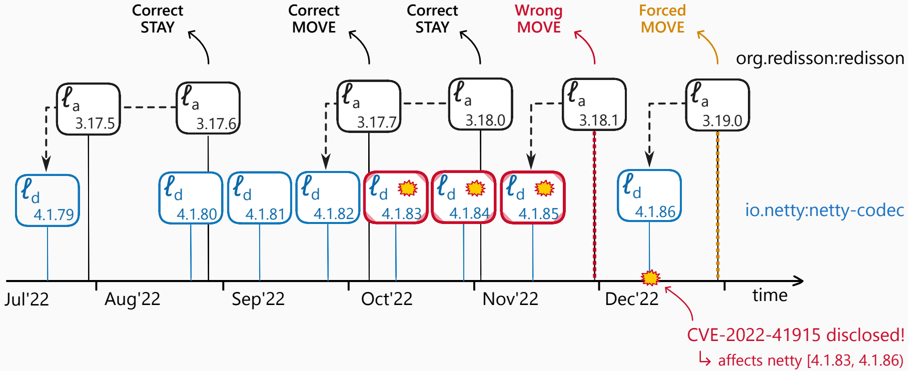
Some motivation (plz!)



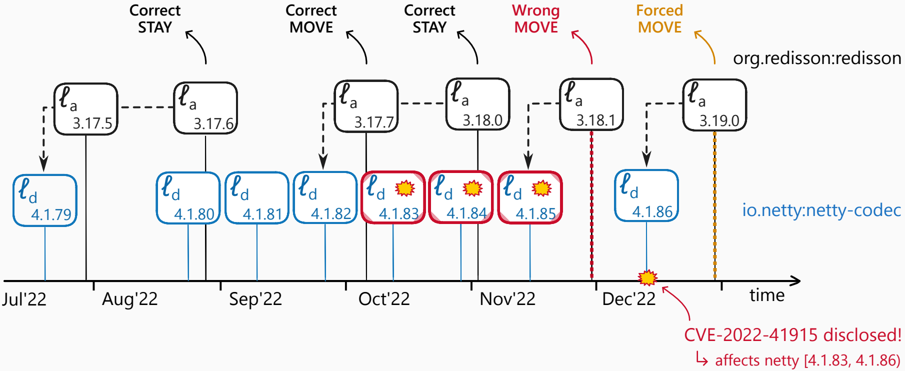
Some motivation (plz!)



Some motivation (plz!)

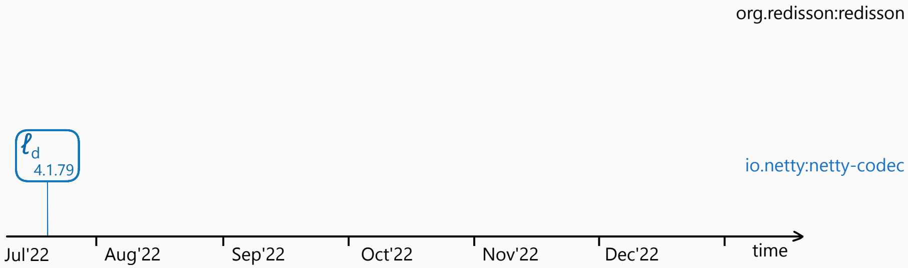


Hindsight!


Correct
STAY
Correct
MOVE
Correct
STAY
Wrong
MOVE
Forced
MOVE
org.redisson:redisson
ℓa 3.17.5
ℓa 3.17.6
ℓa 3.17.7
ℓa 3.18.0
ℓa 3.18.1
ℓa 3.19.0
ℓd 4.1.79
ℓd 4.1.80
ℓd 4.1.81
ℓd 4.1.82
ℓd 4.1.83
ℓd 4.1.84
ℓd 4.1.85
ℓd 4.1.86
io.netty:netty-codec
Jul'22
Aug'22
Sep'22
Oct'22
Nov'22
Dec'22
time
CVE-2022-41915 disclosed!
↳ affects netty [4.1.83, 4.1.86)

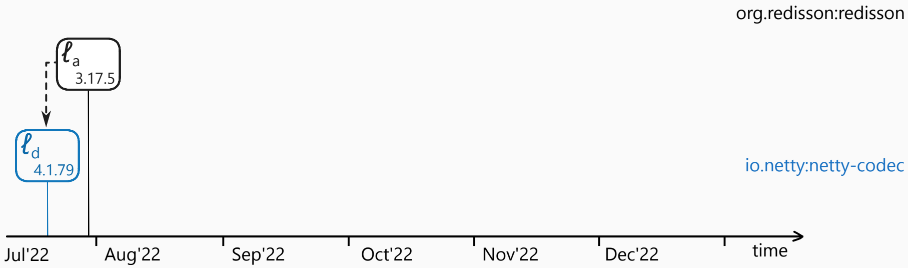
Some motivation (plz!)

Developer perspective in time:

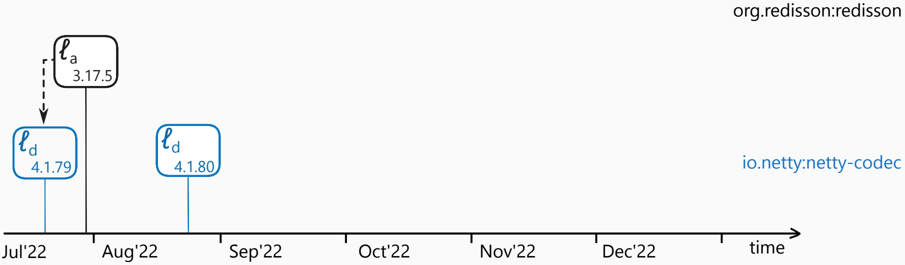


Some motivation (plz!)

Developer perspective in time:

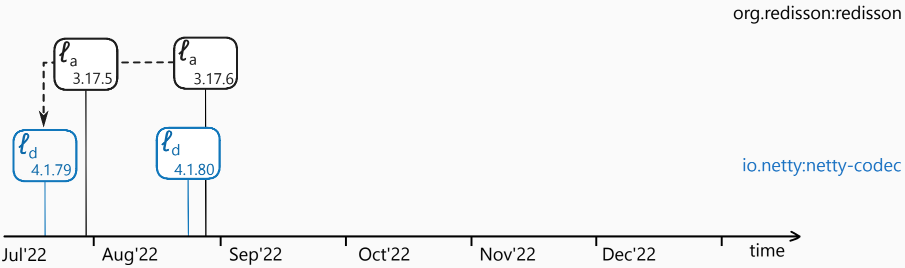


Developer perspective in time:



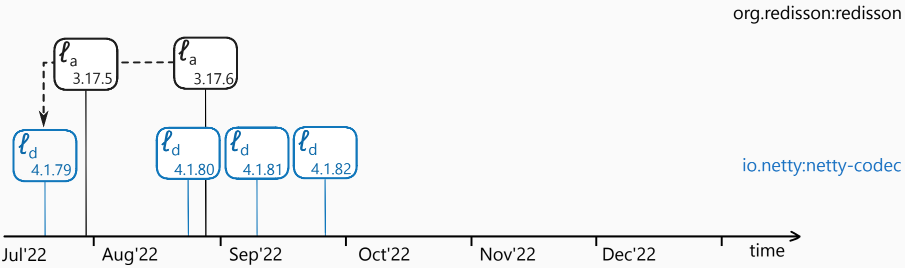
Some motivation (plz!)

Developer perspective in time:



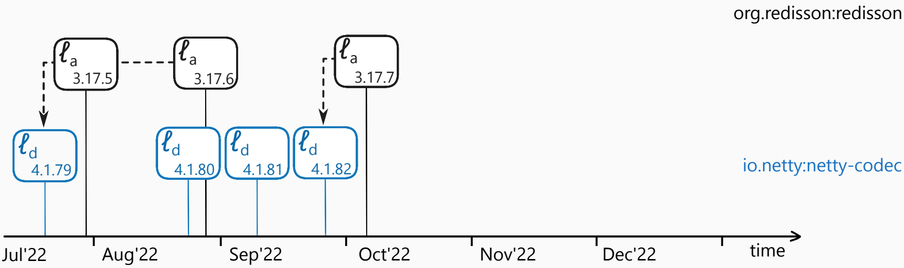
Some motivation (plz!)

Developer perspective in time:

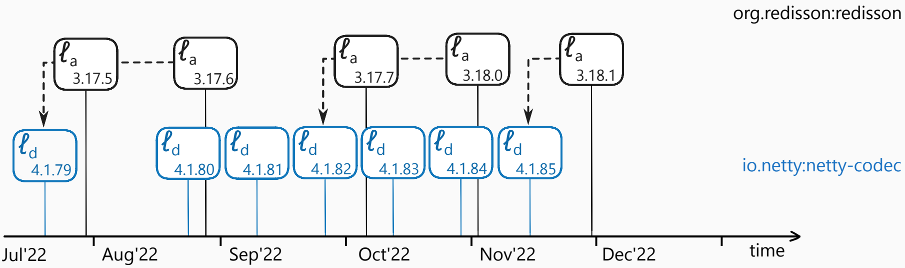


Some motivation (plz!)

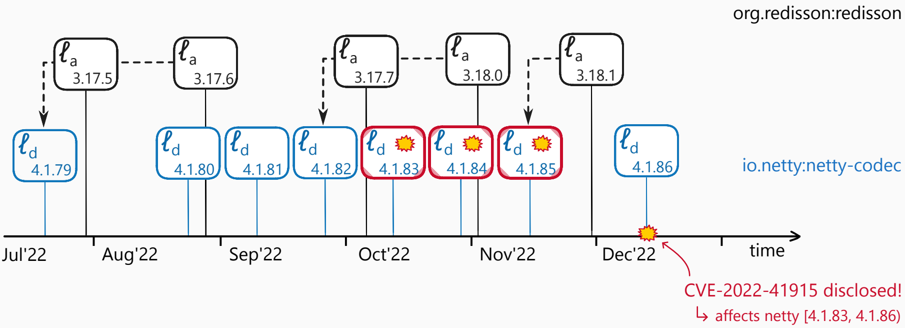
Developer perspective in time:



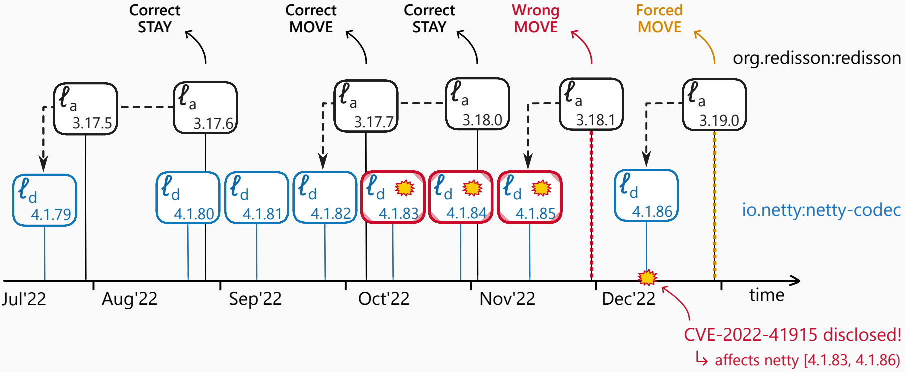
Developer perspective in time:



Developer perspective in time:

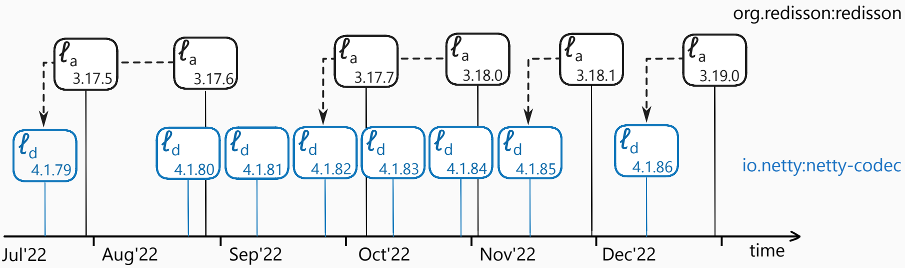


Developer perspective in time:



Some motivation (plz!)

Developer perspective in time:



Is there a best time to update?

Questions

Q1 How does time affect the $\mathsf{Pr}(\text{vuln.})$?

Q2 Which other factors affect $\mathsf{Pr}(\text{vuln.})$?

Questions

Q1 How does time affect the $\Pr(\text{vuln.})$?

- best time to update?

Q2 Which other factors affect $\Pr(\text{vuln.})$?

Questions

Q1 How does time affect the $\Pr(\text{vuln.})$?

- ▷ best time to update?

Q2 Which other factors affect $\Pr(\text{vuln.})$?

- ▷ measurable software metrics

Outta scopes

1. Unpublished/Undetected vulnerabilities:
    - we study publication of CVEs;

Outta scopes

1. Unpublished/Undetected vulnerabilities:
    - we study publication of CVEs;
    - keep it high-level, no code analysis.

Outta scopes

1. Unpublished/Undetected vulnerabilities:
    - we study publication of CVEs;
    - keep it high-level, no code analysis.
2. Probability of *exploitation*:
    - we study publication of CVEs;

Outta scopes

1. Unpublished/Undetected vulnerabilities:
    - we study publication of CVEs;
    - keep it high-level, no code analysis.
2. Probability of *exploitation*:
    - we study publication of CVEs;
    - … but check the work of the EPSS!

Background

Work	Goal		Data				Method			Approach			Projects/Libs.		
	Disc.	Pred.	CVEs	Code	VCS	Dep.	Corr.	Clas.	T.Ser.	AH	SA	ML	Language	#	Purport
[4]	✓			✓				✓				✓	C	3	Find vulnerabilities regardless of existent logs such as CVEs (although CWEs may be used). This includes formal methods and static/dynamic code analysis.
[2]	✓				✓		✓	✓				✓	PHP	3	
[16]	✓			✓			✓	✓			✓	✓	Java	4	
[5]	✓			✓	✓			✓		✓			C/C++, PHP, Java, JS, SQL	10	
[11]	✓		✓		✓			✓		✓			C	3	Detect known vulnerabilities (and their correlation to developer activity metrics) from VCS only—e.g. commit churn, peer comments, etc.
[13]	✓		✓		✓		✓			✓			C	1	
[15]	✓		✓		✓		✓			✓	✓		C, ASM	3	
[14]	✓		✓		✓		✓			✓	✓		C, ASM	1	
[6]	✓		✓	✓				✓				✓	C/C++	3	Detect known vulnerabilities (and their correlation to code metrics) from code only—e.g. number of classes, code cloning, cyclomatic complexity, etc.
[8]	✓		✓	✓				✓				✓	Java	7	
[23]	✓		✓	✓			✓	✓			✓	✓	Java	4	
[24]	✓		✓	✓			✓				✓		Java	3	
[25]	✓		✓	✓			✓				✓		Java	5	
[21]	✓		✓	✓				✓		✓			C	7	
[1]	✓		✓	✓	✓		✓	✓				✓	C/C++	>150k	Detect known vulnerabilities (and their corr. to code and developer activity metrics) from both code and VCS, but without considering the effect of dependencies in their propagation.
[9]	✓		✓	✓	✓			✓		✓			C/C++	8	
[3]	✓		✓	✓	✓		✓				✓		C/C++	5	
[7]	✓		✓	✓	✓		✓	✓			✓	✓	C/C++, Java	1	
[22]	✓		✓	✓	✓		✓				✓	✓	C/C++	2	
[18]	✓		✓	✓	✓	✓		✓		✓			Java	500	Detect known vulnerabilities using code or VCS, via dependency-aware models that can find the offending code to help correcting it (own vs. third-party libraries).
[12]	✓		✓	✓		✓		✓				✓	Java	>300k	
[19]	✓		✓	✓	✓	✓	✓	✓			✓		Java, Ruby, Python	450	
[17]	✓		✓	✓		✓		✓		✓			Java	200	
[26]		✓	✓						✓		✓	✓	Agnostic	9	Time regression to predict vulnerabilities from NVD logs, but the models lack data from the security domain.
[10]		✓	✓						✓		✓	✓	Agnostic	25	
[20]		✓	✓						✓		✓		Agnostic	5	

Work	Goal		Data				Method			Approach			Projects/Libs.		Purport
	Disc.	Pred.	CVEs	Code	VCS	Dep.	Corr.	Clas.	T.Ser.	AH	SA	ML	Language	#	
[4]	✓			✓				✓				✓	C	3	Find vulnerabilities regardless of existent logs such as CVEs (although CWEs may be used). This includes formal methods and static/dynamic code analysis.
[2]	✓				✓		✓	✓				✓	PHP	3	
[16]	✓			✓			✓	✓			✓	✓	Java	4	
[5]	✓			✓	✓			✓		✓			C/C++, PHP, Java, JS, SQL	10	
[11]	✓		✓		✓			✓		✓			C	3	Detect known vulnerabilities (and their correlation to developer activity metrics) from VCS only—e.g. commit churn, peer comments, etc.
[13]	✓		✓		✓		✓			✓			C	1	
[15]	✓		✓		✓		✓			✓	✓		C, ASM	3	
[14]	✓		✓		✓		✓			✓	✓		C, ASM	1	
[6]	✓		✓	✓				✓				✓	C/C++	3	Detect known vulnerabilities (and their correlation to code metrics) from code only—e.g. number of classes, code cloning, cyclomatic complexity, etc.
[8]	✓		✓	✓				✓				✓	Java	7	
[23]	✓		✓	✓			✓	✓			✓	✓	Java	4	
[24]	✓		✓	✓			✓				✓		Java	3	
[25]	✓		✓	✓			✓				✓		Java	5	
[21]	✓		✓	✓				✓		✓			C	7	
[1]	✓		✓	✓	✓		✓	✓				✓	C/C++	>150k	Detect known vulnerabilities (and their corr. to code and developer activity metrics) from both code and VCS, but without considering the effect of dependencies in their propagation.
[9]	✓		✓	✓	✓			✓		✓			C/C++	8	
[3]	✓		✓	✓	✓		✓				✓		C/C++	5	
[7]	✓		✓	✓	✓		✓	✓			✓	✓	C/C++, Java	1	
[22]	✓		✓	✓	✓		✓				✓	✓	C/C++	2	
[18]	✓		✓	✓	✓	✓		✓		✓			Java	500	Detect known vulnerabilities using code or VCS, via dependency-aware models that can find the offending code to help correcting it (own vs. third-party libraries).
[12]	✓		✓	✓		✓		✓				✓	Java	>300k	
[19]	✓		✓	✓	✓	✓	✓	✓			✓		Java, Ruby, Python	450	
[17]	✓		✓	✓		✓		✓		✓			Java	200	
[26]		✓	✓						✓		✓	✓	Agnostic	9	Time regression to predict vulnerabilities from NVD logs, but the models lack data from the security domain.
[10]		✓	✓						✓		✓	✓	Agnostic	25	
[20]		✓	✓						✓		✓		Agnostic	5	

Most works try to discover current vulnerabilities, not predict future ones

Most works disregard the code dependency tree

Most works do not consider time in their analyses

Disregarded security data

Q2 $\Pr(\mathrm{vuln.})$ as function of software metrics

Q1 $\Pr(\mathrm{vuln.})$ as function of time

Q2 $\Pr(\text{vuln.})$ as function of software metrics

- ML & statistical analysis to correlate SE metrics to existent vulnerabilities

Q1 $\Pr(\text{vuln.})$ as function of time

Q2 $\Pr(\text{vuln.})$ as function of software metrics

- ML & statistical analysis to correlate SE metrics to existent vulnerabilities
- human-in-the-loop metrics, including VCS (#commits, seniority...)

Q1 $\Pr(\text{vuln.})$ as function of time

Q2 $\Pr(\text{vuln.})$ as function of software metrics

- ML & statistical analysis to correlate SE metrics to existent vulnerabilities
- human-in-the-loop metrics, including VCS (#commits, seniority...)
- (a few) considerations of own and 3rd party dependencies

Q1 $\Pr(\text{vuln.})$ as function of time

Q2 $\Pr(\text{vuln.})$ as function of software metrics

- ML & statistical analysis to correlate SE metrics to existent vulnerabilities
- human-in-the-loop metrics, including VCS (#commits, seniority…)
- (a few) considerations of own and 3$^{\text{rd}}$ party dependencies

Q1 $\Pr(\text{vuln.})$ as function of time

- time-regression models on CVE publications ($\approx$ FinTech)

Gap analysis

- Studies typically try to *detect*, not *foretell* vulnerabilities.

Gap analysis

- Studies typically try to *detect*, not *foretell* vulnerabilities.
- The dependency tree is seldom analysed (own code only).

Gap analysis

- Studies typically try to *detect*, not *foretell* vulnerabilities.
- The dependency tree is seldom analysed (own code only).
- The rare-event nature of vulnerabilities is disregarded.

Gap analysis

- Studies typically try to *detect*, not *foretell* vulnerabilities.
- The dependency tree is seldom analysed (own code only).
- The rare-event nature of vulnerabilities is disregarded.

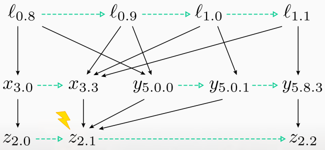
We propose white-box model(s) to fill these gaps

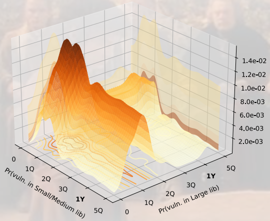
Forecast model

Forecast model

1. Introduction
2. Background
3. Forecast model
4. Conclusions

Time Dependency Trees



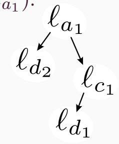


CVE root-lib PDFs

Time Dependency Trees

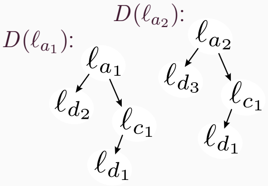
Dependency Trees in time

$D(\ell_{a_1})$:



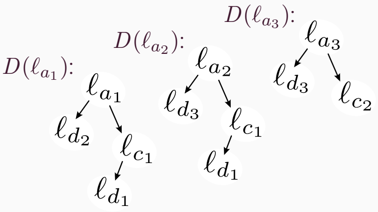
Time Dependency Trees

Dependency Trees in time



Time Dependency Trees

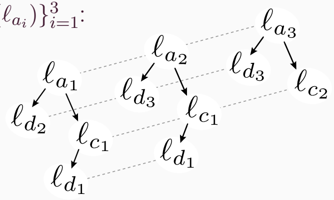
Dependency Trees in time



Time Dependency Trees

Dependency Trees in time

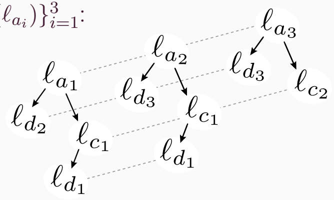
$\{D(\ell_{a_i})\}_{i=1}^{3}$:



Time Dependency Trees

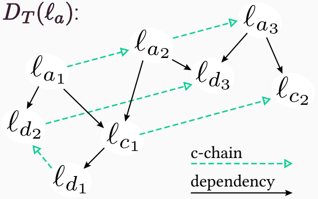
Dependency Trees in time

$\{D(\ell_{a_i})\}_{i=1}^{3}$:



Time Dependency Tree

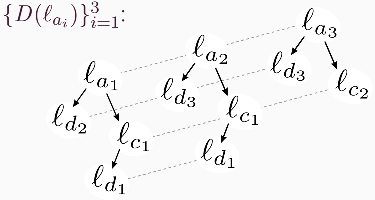
$D_T(\ell_a)$:

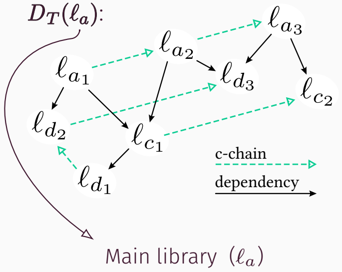


Time Dependency Trees

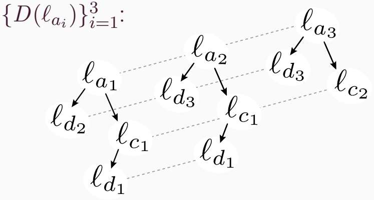
Dependency Trees in time

Time Dependency Tree

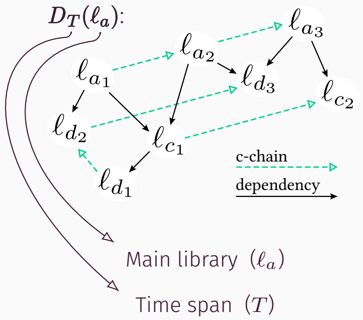




Dependency Trees in time

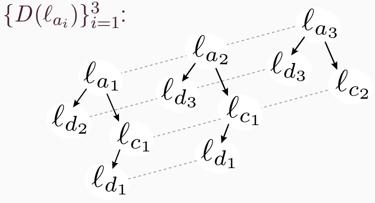


Time Dependency Tree



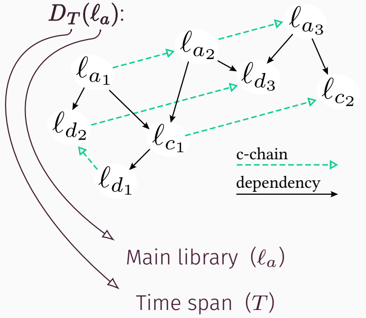
Time Dependency Trees

Dependency Trees in time



$D_t(\ell_a) = D(\ell_{a_1})$
for any time point $t \in T$
after the release of ℓ_{a_1} and
before the release of ℓ_{a_2}

Time Dependency Tree



Properties of TDT $D_T(\ell)$

- Minimal graph representation (no lib-version repetition)

Properties of TDT $D_T(\ell)$

- Minimal graph representation (no lib-version repetition)
- Canonical for library ℓ and time span T

Properties of TDT $D_T(\ell)$

- Minimal graph representation (no lib-version repetition)
- Canonical for library ℓ and time span T
- Natural lifting of dependency trees to time

Properties of TDT $D_T(\ell)$

Theoretical

- Minimal graph representation (no lib-version repetition)
- Canonical for library ℓ and time span T
- Natural lifting of dependency trees to time

Properties of TDT $D_T(\ell)$

Theoretical

- Minimal graph representation (no lib-version repetition)
- Canonical for library ℓ and time span T
- Natural lifting of dependency trees to time

Practical

- **Time-indexing** $D_t(\ell)$ yields the dep. tree at time $t \in T$

Properties of TDT $D_T(\ell)$

Theoretical

- Minimal graph representation (no lib-version repetition)
- Canonical for library ℓ and time span T
- Natural lifting of dependency trees to time

Practical

- **Time-indexing** $D_t(\ell)$ yields the dep. tree at time $t \in T$
- **Library-slicing** $D_T(\ell)\big|_d$ yields *all instances* of dependency d during time T

Properties of TDT $D_T(\ell)$

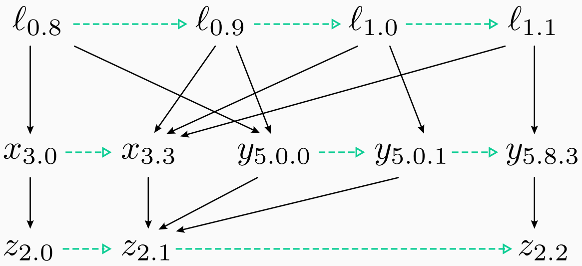
Theoretical

- Minimal graph representation (no lib-version repetition)
- Canonical for library ℓ and time span T
- Natural lifting of dependency trees to time

Practical

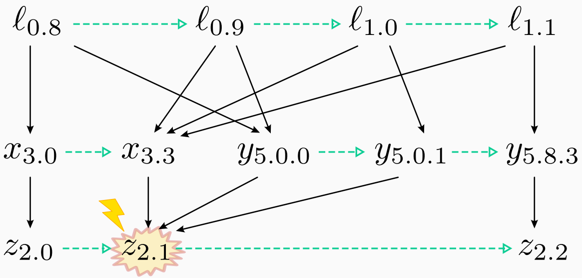
- **Time-indexing** $D_t(\ell)$ yields the dep. tree at time $t \in T$
- **Library-slicing** $D_T(\ell)\big|_d$ yields *all instances* of dependency d during time T
- Reachability analysis can spot single-points-of-failure

My personal project uses $\ell_{1.0}$



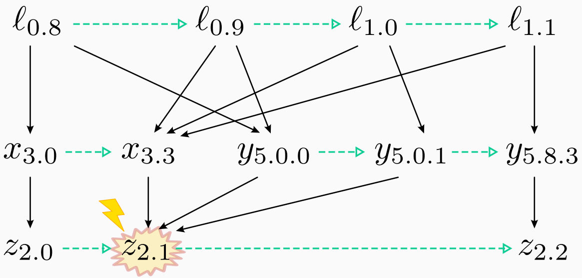
SPoF in time and dependencies

My personal project uses $\ell_{1.0}$



SPoF in time and dependencies

My personal project uses $\ell_{1.0}$



Should I downgrade to $\ell_{0.9}$ or upgrade to $\ell_{1.1}$?

Properties of TDT $D_T(\ell)$

Theoretical

- Minimal graph representation (no lib-version repetition)
- Canonical for library ℓ and time span T
- Natural lifting of dependency trees to time

Practical

- **Time-indexing** $D_t(\ell)$ yields the dep. tree at time $t \in T$
- **Library-slicing** $D_T(\ell)\big|_d$ yields *all instances* of dependency d during time T
- Reachability analysis can spot single-points-of-failure

Properties of TDT $D_T(\ell)$

Theoretical

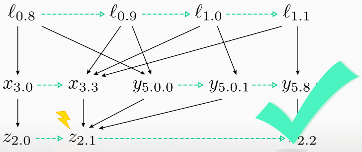
- Minimal graph representation (no lib-version repetition)
- Canonical for library ℓ and time span T
- Natural lifting of dependency trees to time

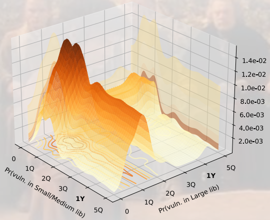
Practical

- **Time-indexing** $D_t(\ell)$ yields the dep. tree at time $t \in T$
- **Library-slicing** $D_T(\ell)\big|_d$ yields *all instances* of dependency d during time T
- Reachability analysis can spot single-points-of-failure
- Can measure health/risk of development environment

Forecast model

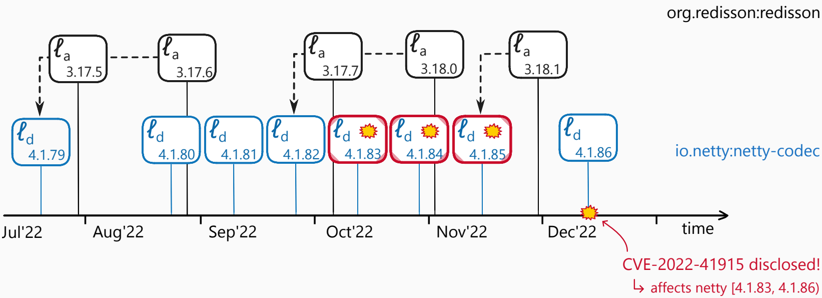
Time Dependency Trees



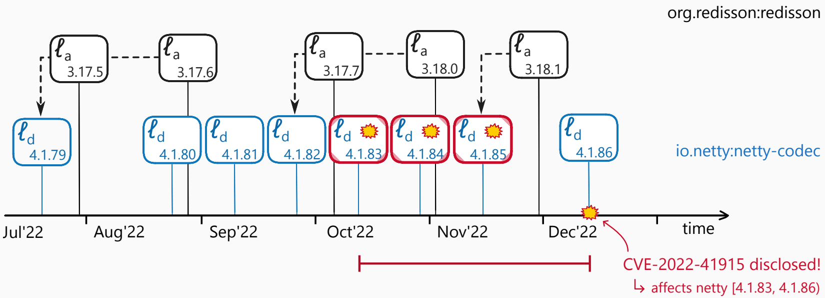


CVE root-lib PDFs

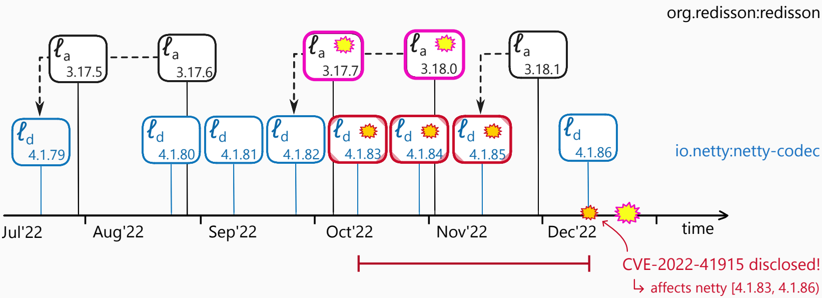
Publication of CVE since time of code release



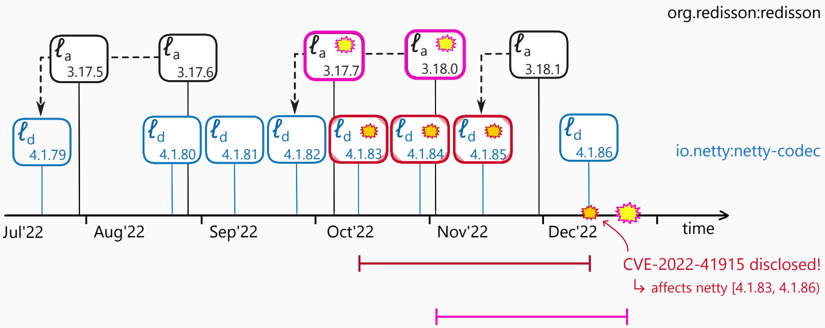
Publication of CVE since time of code release



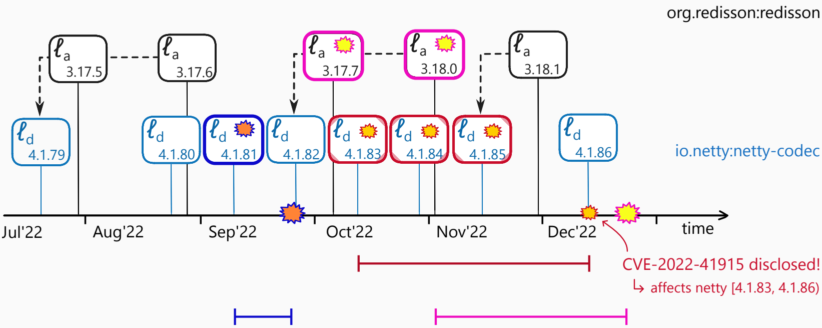
Publication of CVE since time of code release



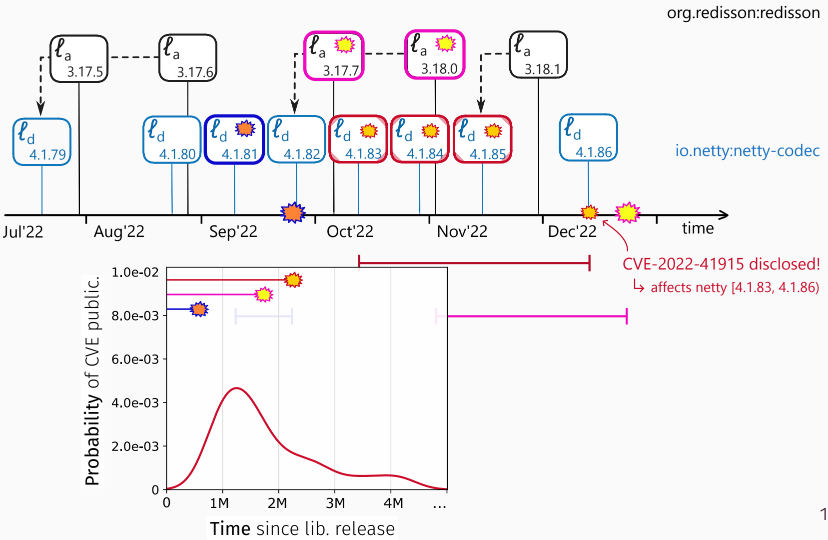
Publication of CVE since time of code release



Publication of CVE since time of code release



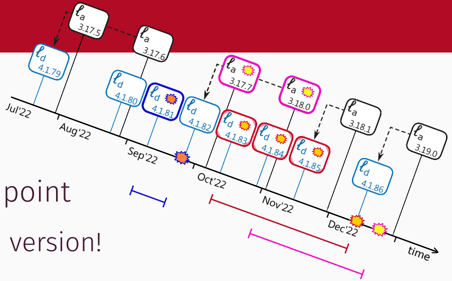
Publication of CVE since time of code release



Rules of the game

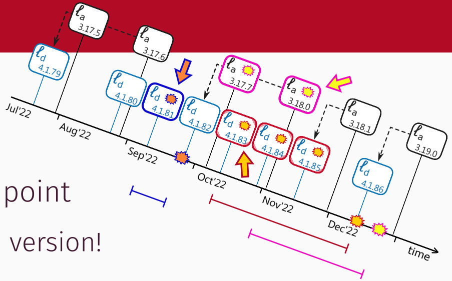
- Count each CVE as one data point
  - must choose one affected version!

Rules of the game



- Count each CVE as one data point
  - must choose one affected version!

Rules of the game



- Count each CVE as one data point
  - must <u>choose one</u> affected version!

Rules of the game

- Count each CVE as one data point
  - must choose one affected version!
- Discriminate per development environment
  - e.g. Java and C/C++ have different vuln. (and times!)

Rules of the game

- Count each CVE as one data point
  - must choose one affected version!
- Discriminate per development environment
  - e.g. Java and C/C++ have different vuln. (and times!)

    🍏 🍊

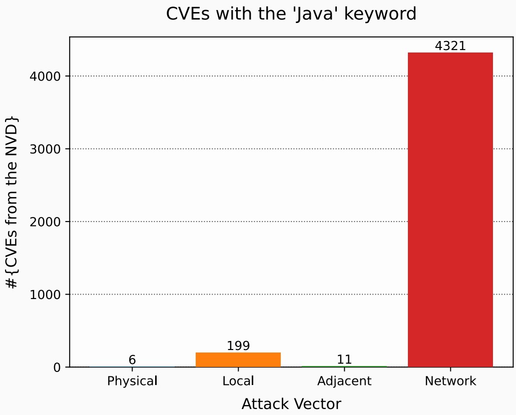
Rules of the game

- Count each CVE as one data point
  - must choose one affected version!
- Discriminate per development environment
  - e.g. Java and C/C++ have different vuln. (and times!)
- Discriminate per library type
  - consider security-relevant code metrics

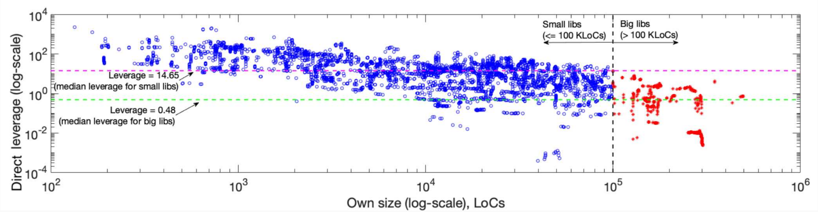
Rules of the game

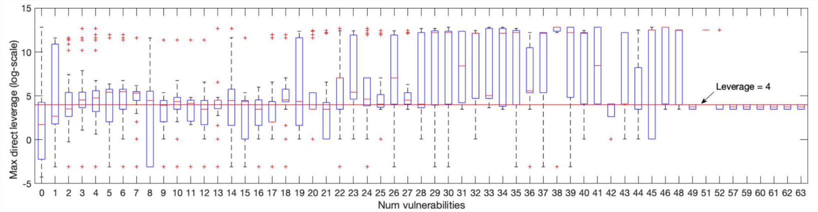
- Count each CVE as one data point
  - must choose one affected version!
- Discriminate per development environment
  - e.g. Java and C/C++ have different vuln. (and times!)
- Discriminate per library type
  - consider security-relevant code metrics 🤌

Used in remote networks



(Own) Code size





Security-relevant code metrics

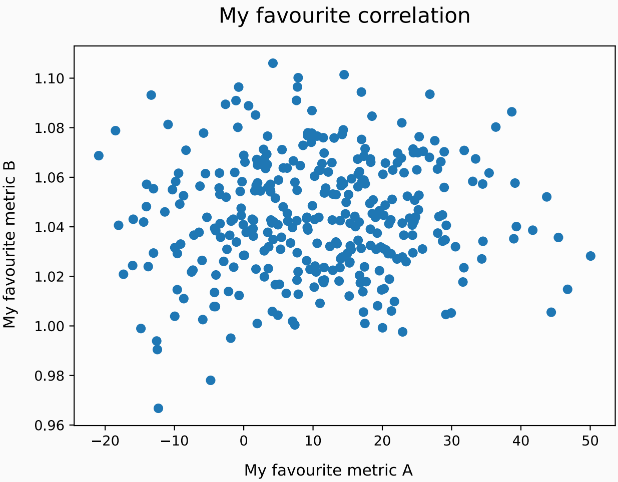
Work	Goal: Disc.	Goal: Pred.	Data: CVEs	Data: Code	Data: VCS	Data: Dep.	Method: Corr.	Method: Clas.	Method: T.Ser.	Approach: AH	Approach: SA	Approach: ML	Projects/Libs.: Language	Projects/Libs.: #	Purport
[4]	✓			✓				✓				✓	C	3	Find vulnerabilities regardless of existent logs such as CVEs (although CWEs may be used). This includes formal methods and static/dynamic code analysis.
[2]	✓				✓		✓	✓				✓	PHP	3	
[16]	✓			✓			✓	✓			✓	✓	Java	4	
[5]	✓			✓	✓			✓		✓			C/C++, PHP, Java, JS, SQL	10	
[11]	✓		✓		✓			✓		✓			C	3	Detect known vulnerabilities (and their correlation to developer activity metrics) from VCS only—e.g. commit churn, peer comments, etc.
[13]	✓		✓		✓		✓			✓			C	1	
[15]	✓		✓		✓		✓			✓	✓		C, ASM	3	
[14]	✓		✓		✓		✓			✓	✓		C, ASM	1	
[6]	✓		✓	✓				✓				✓	C/C++	3	Detect known vulnerabilities (and their correlation to code metrics) from code only—e.g. number of classes, code cloning, cyclomatic complexity, etc.
[8]	✓		✓	✓				✓				✓	Java	7	
[23]	✓		✓	✓			✓	✓			✓	✓	Java	4	
[24]	✓		✓	✓			✓				✓		Java	3	
[25]	✓		✓	✓			✓				✓		Java	5	
[21]	✓		✓	✓				✓		✓			C	7	
[1]	✓		✓	✓	✓		✓	✓				✓	C/C++	>150k	Detect known vulnerabilities (and their corr. to code and developer activity metrics) from both code and VCS, but without considering the effect of dependencies in their propagation.
[9]	✓		✓	✓	✓			✓		✓			C/C++	8	
[3]	✓		✓	✓	✓		✓				✓		C/C++	5	
[7]	✓		✓	✓	✓		✓	✓			✓	✓	C/C++, Java	1	
[22]	✓		✓	✓	✓		✓				✓	✓	C/C++	2	
[18]	✓		✓	✓	✓	✓		✓		✓			Java	500	Detect known vulnerabilities using code or VCS, via dependency-aware models that can find the offending code to help correcting it (own vs. third-party libraries).
[12]	✓		✓	✓		✓		✓				✓	Java	>300k	
[19]	✓		✓	✓	✓	✓	✓	✓			✓		Java, Ruby, Python	450	
[17]	✓		✓	✓		✓		✓		✓			Java	200	
[26]		✓	✓						✓		✓	✓	Agnostic	9	Time regression to predict vulnerabilities from NVD logs, but the models lack data from the security domain.
[10]		✓	✓						✓		✓	✓	Agnostic	25	
[20]		✓	✓						✓		✓		Agnostic	5	

Security-relevant code metrics

Used in remote networks

Own-code size

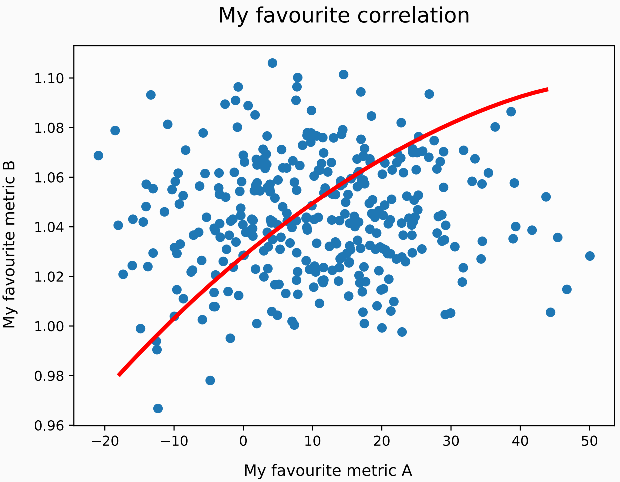
Security-relevant code metrics



Used in remote networks

Own-code size

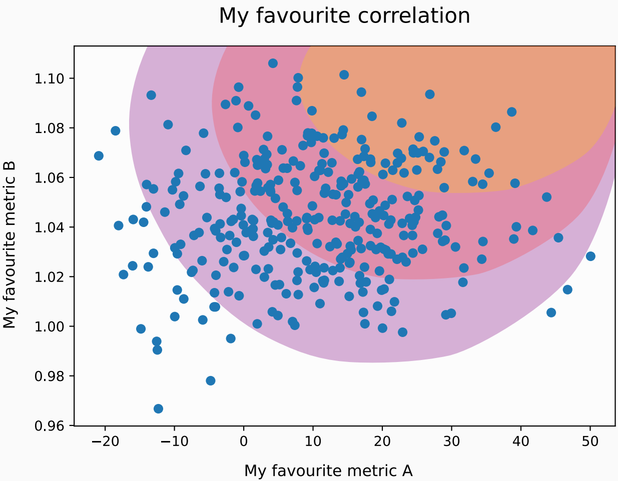
Security-relevant code metrics



Used in remote networks

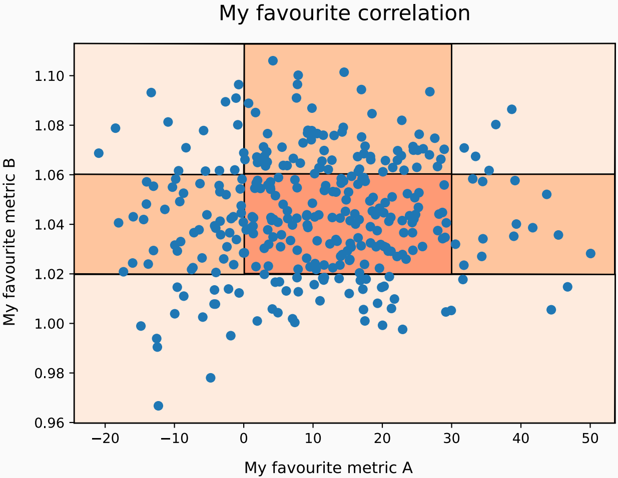
Own-code size

Security-relevant code metrics



Used in remote networks

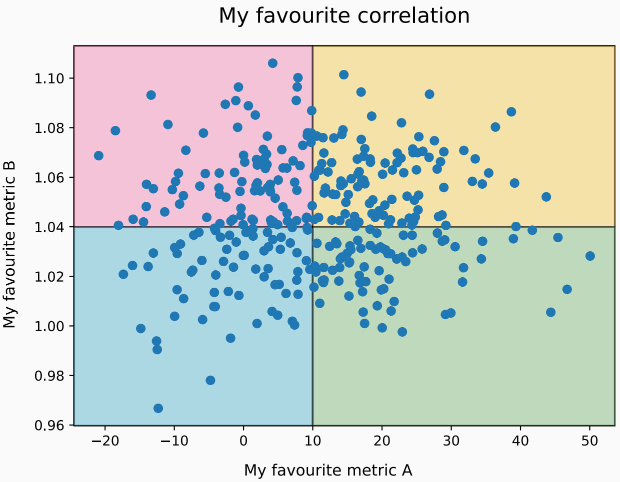
Own-code size



Used in remote networks

Own-code size

Security-relevant code metrics

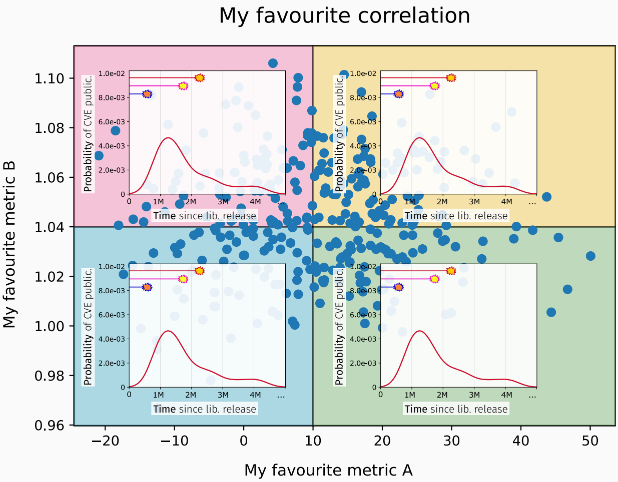


Used in remote networks

Own-code size

Security-relevant code metrics

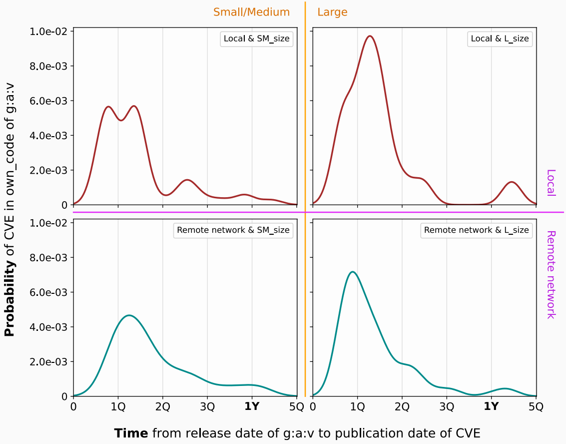
Used in remote networks



Own-code size

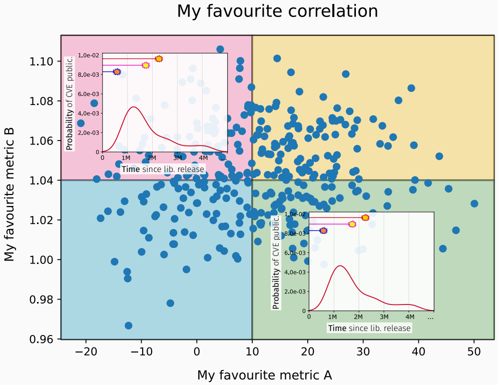
Security-relevant code metrics

Used in remote networks

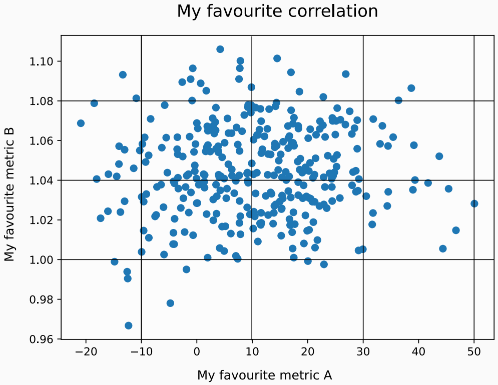


Own-code size

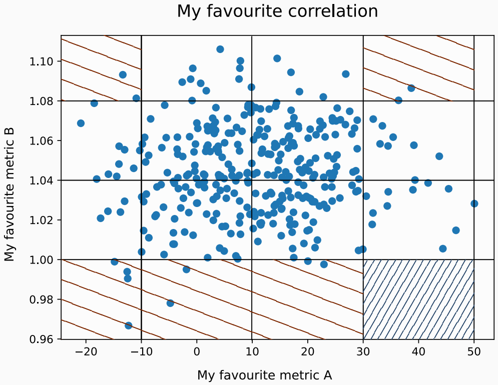
On overfitting and rare events



On overfitting and rare events



On overfitting and rare events



On overfitting and rare events

- Count each CVE as one data point
- Discriminate per development environment
- Discriminate per library type

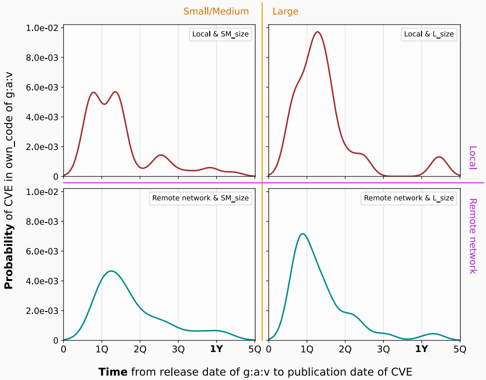
On overfitting and rare events

- Count each CVE as one data point
- Discriminate per development environment
- Discriminate per library type
- Clusterisation mustn't be too thin
  - few divisions per metric-dimension
  - few metric-dimensions

Enough!

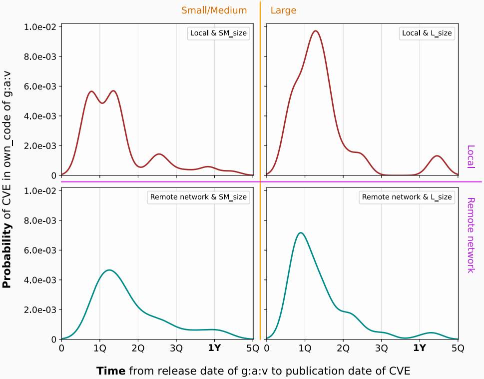
Gimme results

Here ya go



Small/Medium
Large
Local & SM_size
Local & L_size
Remote network & SM_size
Remote network & L_size
Local
Remote network
1.0e-02
8.0e-03
6.0e-03
4.0e-03
2.0e-03
0
0
1Q
2Q
3Q
1Y
5Q
Probability of CVE in own_code of g:a:v
Time from release date of g:a:v to publication date of CVE

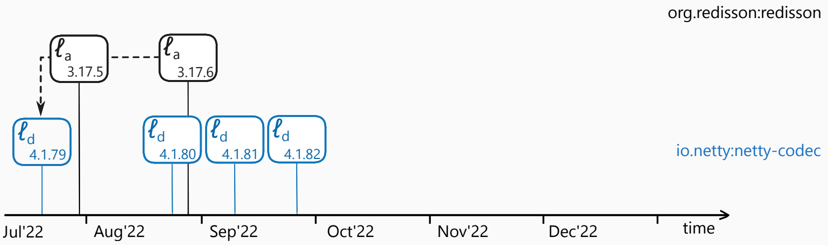
Here ya go



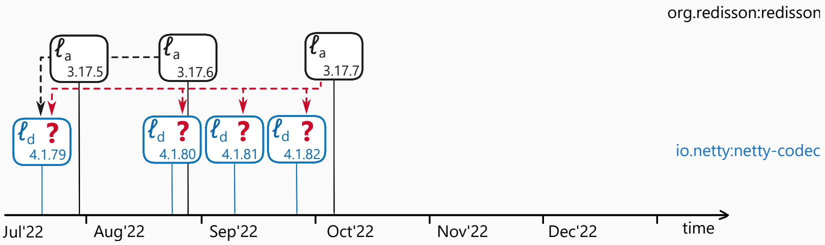
Q1 $\Pr(\text{vuln.})$ as function of time

Q2 $\Pr(\text{vuln.})$ as function of software metrics

Survival analysis on library update



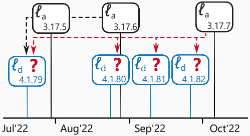
Survival analysis on library update



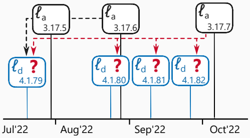
Survival analysis on library update

$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 ("today").

- ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$



Survival analysis on library update

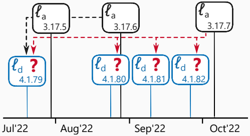


$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 ("today").

- ▷ ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$

Q: $\mathsf{Pr}_{A,B}(t)$ = probability of vuln. of $A \xrightarrow{t} B$ as a function of t

Survival analysis on library update



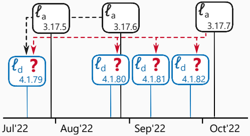
$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 ("today").

- ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$

Q: $\mathrm{Pr}_{A,B}(t)$ = probability of vuln. of $A \xrightarrow{t} B$ as a function of t

A: $\mathrm{Pr}_{A,B}(t) = 1 - \mathrm{SF}_A\big(t + \Delta t_A\big)\,\mathrm{CDF}_B\big(t + \Delta t_B\big)$ where $\Delta t_x \doteq |t_x - t_0|$

Survival analysis on library update



$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 ("today").

- ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$

Q: $\mathsf{Pr}_{A,B}(t) =$ probability of vuln. of $A \xrightarrow{t} B$ as a function of t

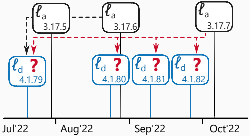
A: $\mathsf{Pr}_{A,B}(t) = 1 - \mathsf{SF}_A(t + \Delta t_A)\,\mathsf{CDF}_B(t + \Delta t_B)$ where $\Delta t_x \doteq |t_x - t_0|$

vuln. in ℓ_A before change vuln. in ℓ_B after change

Survival analysis on library update

$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 (“today”).

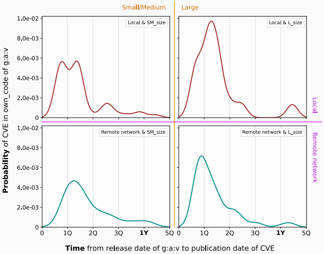
- ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$



Q: $\text{Pr}_{A,B}(t) =$ probability of vuln. of $A \xrightarrow{t} B$ as a function of t

A: $\text{Pr}_{A,B}(t) = 1 - \text{SF}_A(t + \Delta t_A)\ \text{CDF}_B(t + \Delta t_B)$ where $\Delta t_x \doteq |t_x - t_0|$

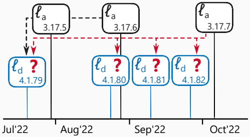
vuln. in ℓ_A before change   vuln. in ℓ_B after change



Survival analysis on library update

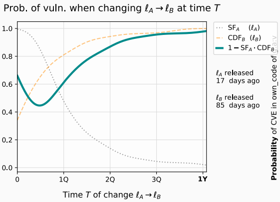
$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 ("today").

- ▷ ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$

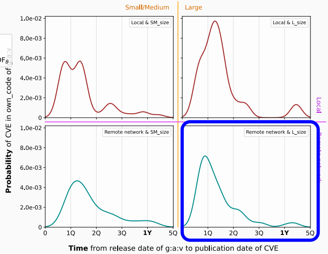


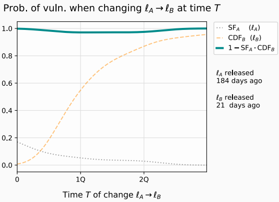
Q: $\mathsf{Pr}_{A,B}(t)$ = probability of vuln. of $A \xrightarrow{t} B$ as a function of t

A: $\mathsf{Pr}_{A,B}(t) = 1 - \mathsf{SF}_A\big(t + \Delta t_A\big)\,\mathsf{CDF}_B\big(t + \Delta t_B\big)$ where $\Delta t_x \doteq |t_x - t_0|$



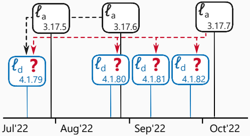
$t_A = 17$ days
$t_B = 85$ days





$t_A = 184$ days
$t_B = 21$ days

Survival analysis on library update

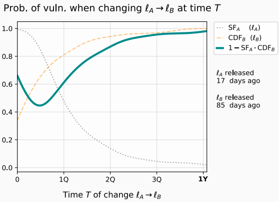


$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 ("today").

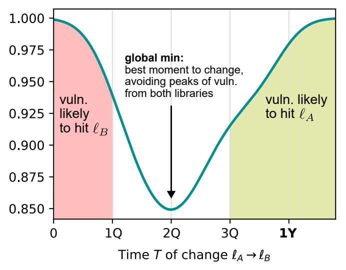
- ▷ ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$

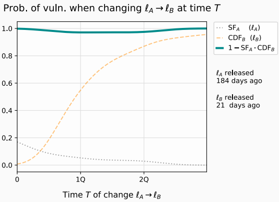
Q: $\mathsf{Pr}_{A,B}(t)$ = probability of vuln. of $A \xrightarrow{t} B$ as a function of t

A: $\mathsf{Pr}_{A,B}(t) = 1 - \mathsf{SF}_A(t + \Delta t_A)\, \mathsf{CDF}_B(t + \Delta t_B)$ where $\Delta t_x \doteq |t_x - t_0|$



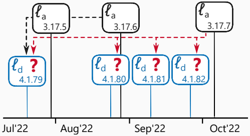
t_A = 17 days
t_B = 85 days





t_A = 184 days
t_B = 21 days

Survival analysis on library update

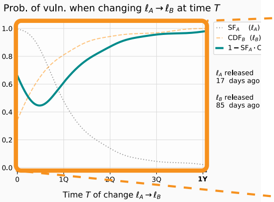


$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 ("today").

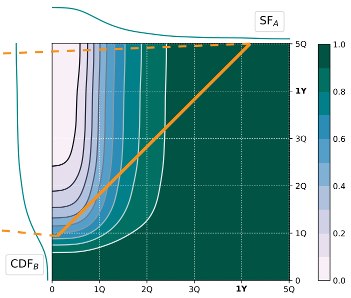
- ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$

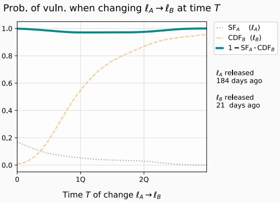
Q: $\mathsf{Pr}_{A,B}(t)$ = probability of vuln. of $A \xrightarrow{t} B$ as a function of t

A: $\mathsf{Pr}_{A,B}(t) = 1 - \mathsf{SF}_A\big(t + \Delta t_A\big)\,\mathsf{CDF}_B\big(t + \Delta t_B\big)$ where $\Delta t_x \doteq |t_x - t_0|$



$t_A = 17$ days
$t_B = 85$ days

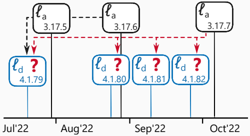




$t_A = 184$ days
$t_B = 21$ days

Survival analysis on library update

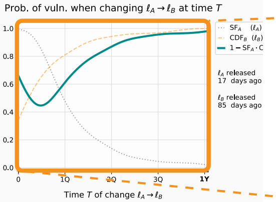
$A \xrightarrow{t} B$ means that we change from dependency ℓ_A to ℓ_B in t time units counting from t_0 ("today").



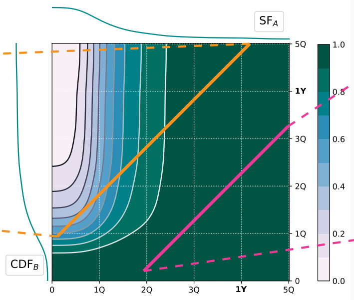
- $\triangleright$ ℓ_A was released on $t_A < t_0$, ℓ_B on $t_B < t_0$, $t_A \bowtie t_B$

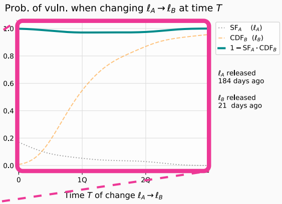
Q: $\mathsf{Pr}_{A,B}(t)$ = probability of vuln. of $A \xrightarrow{t} B$ as a function of t

A: $\mathsf{Pr}_{A,B}(t) = 1 - \mathsf{SF}_A(t + \Delta t_A)\,\mathsf{CDF}_B(t + \Delta t_B)$ where $\Delta t_x \doteq |t_x - t_0|$



t_A = 17 days
t_B = 85 days





t_A = 184 days
t_B = 21 days

Vulnerabilities from any dependency

Q: $\mathsf{Pr}_{A,B}(t) =$ probability of vuln. in ℓ_A or ℓ_B before t

Vulnerabilities from any dependency

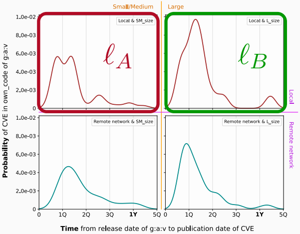
Q: $\mathsf{Pr}_{A,B}(t)$ = probability of vuln. in ℓ_A or ℓ_B before t

A: $\mathsf{Pr}_{A,B}(t) = \mathsf{Pr}(\min(\ell_A, \ell_B) \leqslant t) = 1 - (1 - \mathsf{Pr}_A(t))(1 - \mathsf{Pr}_B(t))$

Vulnerabilities from any dependency

Q: $\mathsf{Pr}_{A,B}(t)$ = probability of vuln. in ℓ_A or ℓ_B before t

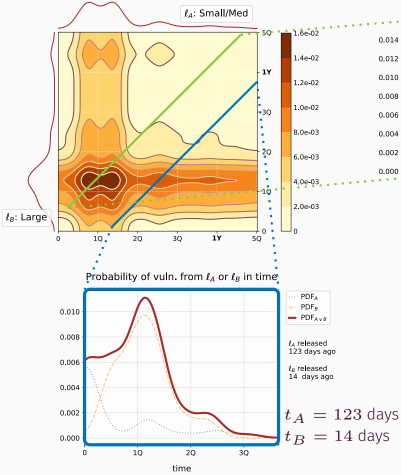
A: $\mathsf{Pr}_{A,B}(t) = \mathsf{Pr}(\min(\ell_A, \ell_B) \leqslant t) = 1 - (1 - \mathsf{Pr}_A(t))(1 - \mathsf{Pr}_B(t))$

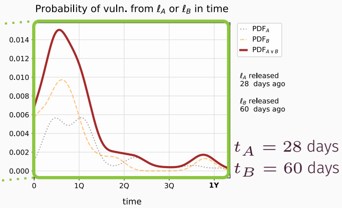


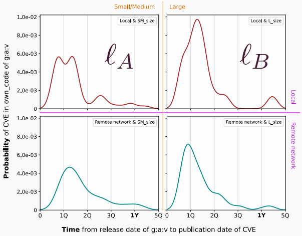
Vulnerabilities from any dependency

Q: $\mathsf{Pr}_{A,B}(t)$ = probability of vuln. in ℓ_A or ℓ_B before t

A: $\mathsf{Pr}_{A,B}(t) = \mathsf{Pr}(\min(\ell_A, \ell_B) \leqslant t) = 1 - (1 - \mathsf{Pr}_A(t))(1 - \mathsf{Pr}_B(t))$



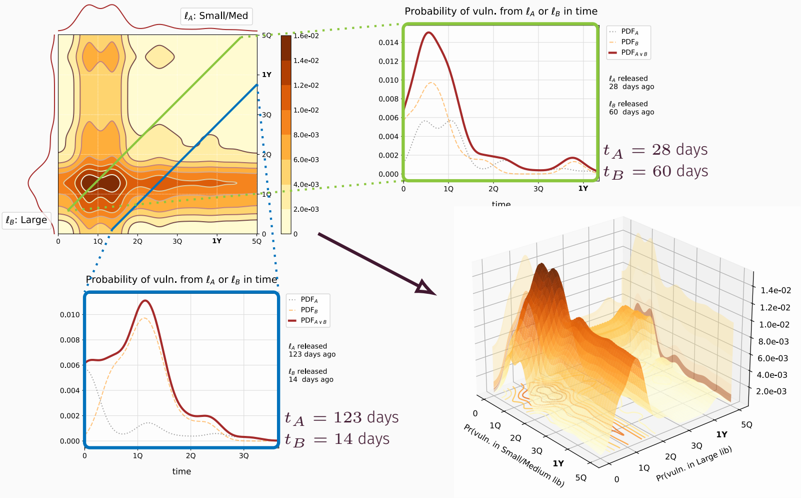




Vulnerabilities from any dependency

Q: $\mathsf{Pr}_{A,B}(t) =$ probability of vuln. in ℓ_A or ℓ_B before t

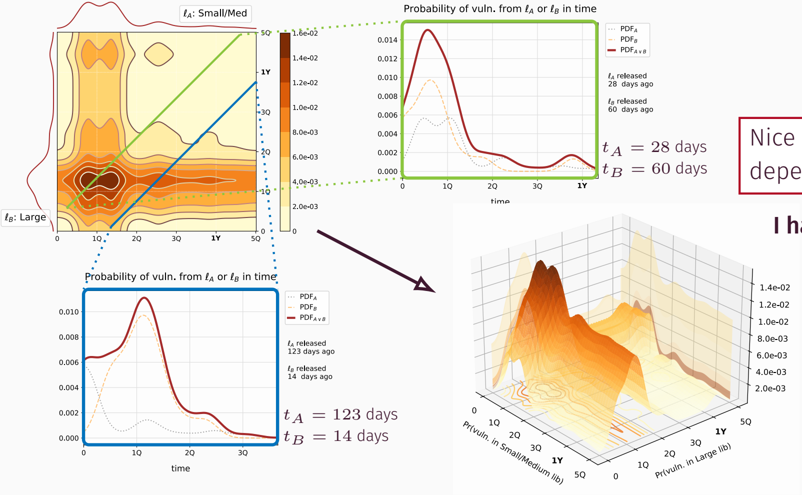
A: $\mathsf{Pr}_{A,B}(t) = \mathsf{Pr}(\min(\ell_A, \ell_B) \leqslant t) = 1 - (1 - \mathsf{Pr}_A(t))(1 - \mathsf{Pr}_B(t))$



Vulnerabilities from any dependency

Q: $\Pr_{A,B}(t)$ = probability of vuln. in ℓ_A or ℓ_B before t

A: $\Pr_{A,B}(t) = \Pr(\min(\ell_A, \ell_B) \leqslant t) = 1 - (1 - \Pr_A(t))(1 - \Pr_B(t))$

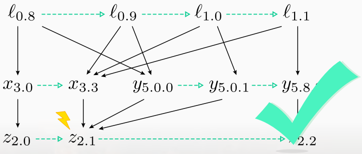


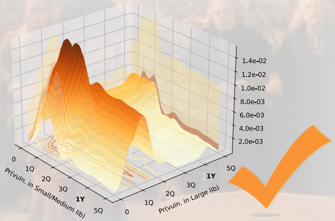
Nice for 2 dependencies...

I have 2000 TDTs!

Forecast model

Time Dependency Trees





CVE root-lib PDFs

Conclusions

Some things done

- Time Dependency Trees

Some things done

- Time Dependency Trees
  - Aggregate dependency and code-evolution data

Some things done

- Time Dependency Trees
  - Aggregate dependency and code-evolution data
  - Minimal representation with nice properties

Some things done

- Time Dependency Trees
  - Aggregate dependency and code-evolution data
  - Minimal representation with nice properties
  - Framework for large-scale project analysis

Some things done

- Time Dependency Trees
  - Aggregate dependency and code-evolution data
  - Minimal representation with nice properties
  - Framework for large-scale project analysis
- Probability of vulnerabilities as a function of time

Some things done

- Time Dependency Trees
  - Aggregate dependency and code-evolution data
  - Minimal representation with nice properties
  - Framework for large-scale project analysis
- Probability of vulnerabilities as a function of time
  - Express time from library release to CVE publication

Some things done

- Time Dependency Trees
  - Aggregate dependency and code-evolution data
  - Minimal representation with nice properties
  - Framework for large-scale project analysis
- Probability of vulnerabilities as a function of time
  - Express time from library release to CVE publication
  - Discriminate per type of library (security-relevant props.)

Some things done

- Time Dependency Trees
  - Aggregate dependency and code-evolution data
  - Minimal representation with nice properties
  - Framework for large-scale project analysis
- Probability of vulnerabilities as a function of time
  - Express time from library release to CVE publication
  - Discriminate per type of library (security-relevant props.)
  - Base information for probability forecasting

Some things done

Some things to be done

Some things done

to be

- Other metrics to clusterise libraries for PDF-fitting

Some things done

to be

- Other metrics to clusterise libraries for PDF-fitting
- Validate in other languages (all Java so far)

Some things ~~done~~ to be done

- Other metrics to clusterise libraries for PDF-fitting
- Validate in other languages (all Java so far)
- SPoF detection—across versions—in Java/Maven

Some things done to be

- Other metrics to clusterise libraries for PDF-fitting
- Validate in other languages (all Java so far)
- SPoF detection—across versions—in Java/Maven
- c-chains polution by CVE

Questions?

J. Akram and P. Luo.
SQVDT: A scalable quantitative vulnerability detection technique for source code security assessment.
Software: Practice and Experience, 51(2):294–318, 2021.

M. Alohaly and H. Takabi.
When do changes induce software vulnerabilities?
In *CIC*, pages 59–66. IEEE, 2017.

H. Alves, B. Fonseca, and N. Antunes.
Software metrics and security vulnerabilities: Dataset and exploratory study.
In *EDCC*, pages 37–44. IEEE, 2016.

Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay.
Vulnerability prediction from source code using machine learning.
IEEE Access, 8:150672–150684, 2020.

A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni.
Identifying the characteristics of vulnerable code changes: An empirical study.
In *FSE*, pages 257–268. ACM, 2014.

References ii

S. Chakraborty, R. Krishna, Y. Ding, and B. Ray.
Deep learning based vulnerability detection: Are we there yet.
IEEE Transactions on Software Engineering, 48(9):3280–3296, 2021.

I. Chowdhury and M. Zulkernine.
Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities.
Journal of Systems Architecture, 57(3):294–313, 2011.

S. Ganesh, T. Ohlsson, and F. Palma.
Predicting security vulnerabilities using source code metrics.
In *SweDS*, pages 1–7. IEEE, 2021.

S. Kim, S. Woo, H. Lee, and H. Oh.
VUDDY: A scalable approach for vulnerable code clone discovery.
In *SP*, pages 595–614. IEEE, 2017.

D. Last.
Forecasting zero-day vulnerabilities.
In *CISRC*, pages 1–4. ACM, 2016.

H. Li, H. Kwon, J. Kwon, and H. Lee.
A scalable approach for vulnerability discovery based on security patches.
In *ATIS*, volume 490 of *CCIS*, pages 109–122. Springer, 2014.

Q. Li, J. Song, D. Tan, H. Wang, and J. Liu.
PDGraph: A large-scale empirical study on project dependency of security vulnerabilities.
In *DSN*, pages 161–173. IEEE, 2021.

A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates.
When a patch goes bad: Exploring the properties of vulnerability-contributing commits.
In *ESEM*, pages 65–74. IEEE, 2013.

A. Meneely and L. Williams.
Secure open source collaboration: An empirical study of Linus' law.
In *CCS*, pages 453––462. ACM, 2009.

References iv

A. Meneely and L. Williams.
Strengthening the empirical analysis of the relationship between Linus' law and software security.
In *ESEM*. ACM, 2010.

Y. Pang, X. Xue, and A. S. Namin.
Predicting vulnerable software components through N-gram analysis and statistical feature selection.
In *ICMLA*, pages 543–548. IEEE, 2015.

I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci.
Vulnerable open source dependencies: Counting those that matter.
In *ESEM*. ACM, 2018.

I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci.
Vuln4Real: A methodology for counting actually vulnerable dependencies.
IEEE Transactions on Software Engineering, 48(5):1592–1609, 2022.

References v

G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo.
Out of sight, out of mind? how vulnerable dependencies affect open-source projects.
Empirical Software Engineering, 26(4), 2021.

Y. Roumani, J. K. Nwankpa, and Y. F. Roumani.
Time series modeling of vulnerabilities.
Computers & Security, 51:32–40, 2015.

N. Shahmehri, A. Mammar, E. Montes de Oca, D. Byers, A. Cavalli, S. Ardi, and W. Jimenez.
An advanced approach for modeling and detecting software vulnerabilities.
Information and Software Technology, 54(9):997–1013, 2012.

Y. Shin, A. Meneely, L. Williams, and J. A. Osborne.
Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities.
IEEE Transactions on Software Engineering, 37(6):772–787, 2011.

References vi

K. Z. Sultana, V. Anu, and T.-Y. Chong.
Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach.
Journal of Software: Evolution and Process, 33(3), 2021.

K. Z. Sultana, A. Deo, and B. J. Williams.
Correlation analysis among Java nano-patterns and software vulnerabilities.
In *HASE*, pages 69–76. IEEE, 2017.

K. Z. Sultana and B. J. Williams.
Evaluating micro patterns and software metrics in vulnerability prediction.
In *SoftwareMining*, pages 40–47. IEEE, 2017.

E. Yasasin, J. Prester, G. Wagner, and G. Schryen.
Forecasting IT security vulnerabilities – an empirical analysis.
Computers & Security, 88, 2020.

Predicting Security Attacks in FOSS

Why you want it and one way to do it

C.E. Budde R. Paramitha F. Massacci
Università di Trento (IT) & Vrije Universiteit (NL)

Vuln4Cast 2023 FIRST Technical Colloquium