ProveNFix: Temporal Property guided Program Repair

Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, Abhik Roychoudhury

17th July @ FSE 2024, Porto de Galinhas, Brazil

Can temporal property analysis be modular?

"Each function is analysed only once and

can be replaced by their verified properties."

Can temporal property analysis be modular?

“Each function is analysed only once and

can be replaced by their verified properties.”

Three main difficulties：

1. Temporal logic property entailment checker.
2. Writing temporal specifications for each function is tedious and challenging.
3. The classic pre/post-conditions is not enough, e.g.,

   “some meaningful operations can only happen if the return value of loading the certificate is positive”

Future-condition

Defined in header `<stdlib.h>`		
`void free( void* ptr );`		

```
void free (void *ptr);
// post: (ptr=null ∧ ϵ) ∨ (ptr≠null ∧ free(ptr))
// future: true ∧ 𝒢 (!_(ptr))
```

The behavior is undefined if after `free()` returns, an access is made through the pointer `ptr` (unless another allocation function happened to result in a pointer value equal to `ptr`).

Defined in header `<stdlib.h>`		
`void* malloc( size_t size );`		

On success, returns the pointer to the beginning of newly allocated memory. To avoid a memory leak, the returned pointer must be deallocated with `free()` or `realloc()`.

On failure, returns a null pointer.

```
void *malloc (size_t size);
// pre: size>0 ∧ _*
// post: (ret=null ∧ ϵ) ∨ (ret≠null ∧ malloc(ret))
// future: ret≠null → ℱ (free(ret))
```

Future-condition based modular analysis

$$\frac{nm(x^*) \mapsto (\Phi_{pre}, \Phi_{post}) \in \mathcal{E} \qquad \boxed{\Phi \sqsubseteq [y^*/x^*]\Phi_{pre}} \qquad \Phi'_{post} = [r/ret, y^*/x^*]\Phi_{post} \qquad \mathcal{E} \vdash \{\Phi \cdot \Phi'_{post}\}\ e\ \{\Phi_e\}}{\mathcal{E} \vdash \{\Phi\}\ r = nm(y^*);\ e\ \{\Phi'_{post} \cdot \Phi_e\}}\ [\mathit{FR\text{-}Call}]$$

Entailment Checking → $\Phi \sqsubseteq [y^*/x^*]\Phi_{pre}$

A collection of specifications → $\mathcal{E}$

Future-condition based modular analysis

Entailment Checking

A collection of specifications

$$\frac{nm(x^*) \mapsto (\Phi_{pre}, \Phi_{post}, \Phi_{future}) \in \mathcal{E} \qquad \Phi \sqsubseteq [y^*/x^*]\Phi_{pre} \qquad \Phi'_{post} = [r/ret, y^*/x^*]\Phi_{post} \qquad \mathcal{E} \vdash \{\Phi \cdot \Phi'_{post}\}\ e\ \{\Phi_e\} \qquad \Phi_e \sqsubseteq [r/ret, y^*/x^*]\Phi_{future}}{\mathcal{E} \vdash \{\Phi\}\ r = nm(y^*);\ e\ \{\Phi'_{post} \cdot \Phi_e\}}\ [FR\text{-}Call]$$

Can temporal property analysis be modular?

"Each function is analysed only once and

can be replaced by their verified properties."

Three main difficulties：

1. Temporal logic property entailment checker.

2. Writing temporal specifications for each function is tedious and challenging.

3. ~~The classic pre/post-conditions is not enough, e.g.,~~ **Future-condition!**

   ~~"some meaningful operations can only happen if the return value of loading the certificate is positive"~~

Specification inference

```
void *malloc (size_t size);
// future: (ret=null ∧ 𝒢 (!_(ret))) ∨ (ret≠null ∧ ℱ (free(ret))
```

```
void wrap_malloc_I (int* ptr)
// future: ptr=null ∧ 𝒢 (!_(ptr))
           ∨ ptr≠null ∧ ℱ (free(ptr))
{ ptr = malloc (4); return;}
```

```
int* wrap_malloc_II ()
// future: ret=null ∧ 𝒢 (!_(ret))
           ∨ ret≠null ∧ ℱ (free(ret))
{ int* ptr = malloc (4); return ptr;}
```

Specification inference

```
void *malloc (size_t size);
// future: (ret=null ∧ 𝒢 (!_(ret))) ∨ (ret≠null ∧ ℱ (free(ret))
```

```
int* wrap_malloc_III ()
// future: true ∧ ℱ (free(ret))
{ int* ptr = malloc (4);
  if (ptr == NULL) exit(-1);
  return ptr;}
```

```
int* wrap_malloc_IV ()
// future: true ∧ _*
{ int* ptr = malloc (4);
 + if (ptr != NULL) free(ptr); // a repair
  return NULL;}
```

Failed entailment: $\mathtt{true} \wedge \mathcal{E} \not\sqsubseteq \mathtt{ptr{\neq}null} \wedge \mathcal{F}\,(\mathtt{free(ptr)})$

Can temporal property analysis be modular?

"Each function is analysed only once and

can be replaced by their verified properties."

Three main difficulties:

1. Temporal logic property entailment checker. **Primitive spec + spec inference!**

~~2. Writing temporal specifications for each function is tedious and challenging.~~

~~3. The classic pre/post-conditions is not enough, e.g.,~~ **Future-condition!**

~~"some meaningful operations can only happen if the return value of loading the certificate is positive"~~

Term rewriting system for regular expressions

- Flexible specifications, which can be combined with other logic;
- Efficient entailment checker with inductive proofs.

$$
\begin{array}{llll}
(\mathit{IntRE}) & \Phi & ::= & \bigvee(\pi \wedge \theta) \\
(\mathit{Traces}) & \theta & ::= & \bot \mid \epsilon \mid \mathrm{I} \mid \theta_1 \cdot \theta_2 \mid \theta_1 \vee \theta_2 \mid \theta^\star \\
(\mathit{Events}) & \mathrm{I} & ::= & \mathrm{A}(v) \mid \mathrm{A}(\_) \mid !\mathrm{A}(v) \mid !\_(v) \mid \_ \mid \mathrm{I}_1 \wedge \mathrm{I}_2 \\
(\mathit{Pure}) & \pi & ::= & T \mid F \mid bop(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \exists x.\pi \\
(\mathit{Terms}) & t & ::= & v \mid t_1 + t_2 \mid t_1 - t_2 \\
(\mathit{Values}) & v & ::= & c \mid x \mid null
\end{array}
$$

Fig. 10. Syntax of the spec language, *IntRE*.

Term rewriting system for regular expressions

- Flexible specifications, which can be combined with other logic;
- Efficient entailment checker with inductive proofs.

Examples:

$x>2 \wedge E \sqsubseteq x>1 \wedge (E \vee F)$

$x>0 \wedge E \not\sqsubseteq x>1 \wedge (E \vee F)$

$true \wedge E \not\sqsubseteq true \wedge (E \,.\, F)$

$$\dfrac{\dfrac{\dfrac{(a \vee b)^{\star} \sqsubseteq (a \vee b \vee bb)^{\star} \quad \text{[Reoccur]}}{\epsilon \cdot (a \vee b)^{\star} \sqsubseteq \epsilon \cdot (a \vee b \vee bb)^{\star}}}{a \cdot (a \vee b)^{\star} \sqsubseteq (a \vee b \vee bb)^{\star}} \qquad \dfrac{\text{[Reoccur]}}{b \cdot (a \vee b)^{\star} \sqsubseteq \ldots}}{(a \vee b)^{\star} \sqsubseteq (a \vee b \vee bb)^{\star}}$$

Can temporal property analysis be modular?

Can!

“Each function is analysed only once and can be replaced by their verified properties.”

Three main difficulties：

A term rewriting system for regular expressions

1. ~~Temporal logic property entailment checker.~~ **Primitive spec + spec inference!**
2. ~~Writing temporal specifications for each function is tedious and challenging.~~
3. ~~The classic pre/post-conditions is not enough, e.g.,~~ **Future-condition!**

   ~~“some meaningful operations can only happen if the return value of loading the certificate is positive”~~

Experiment 1: detecting bugs

Primitive APIs	Pre	Post	Future	Targeted Bug Type
open/socket/fopen/fdopen/opendir	✗	✗	✓	Resource Leak
close/fclose/endmntent/fflush/closedir	✗	✓	✗	
malloc/realloc/calloc/localtime	✗	✗	✓	Null Pointer Dereference
→ *(pointer dereference)*	✗	✓	✗	
malloc	✓	✓	✓	Memory Usage (Leak, Use-After-Free, Double Free)
free	✓	✓	✓	

- 17 predefined primitive specs.
- ProveNFix is finding 72.2% more true bugs, with a 17% loss of missing true bugs.

Project	kLoC	#NPD		#ML		#RL		Time	
		Infer	ProveNFix	Infer	ProveNFix	Infer	ProveNFix	Infer	ProveNFix
Swoole(a4256e4)	44.5	30+7	30+23	16+4	12+16	13+1	13+6	2m 50s	39.54s
lxc(72cc48f)	63.3	7+9	5+19	11+6	10+12	5+1	5+5	55.62s	1m 28s
WavPack(22977b2)	36	23+7	20+21	3	3+9	0+2	0	27.99s	23.77s
flex(d3de49f)	23.9	14+4	14+4	3	3+1	0	0+1	32.25s	47.75s
p11-kit	76.2	3+5	2+2	13+3	12+15	5	5+1	1m 57s	1m 4s
x264(d4099dd)	67.7	0	0	12	11+5	2	2+3	2m 33s	23.168s
recutils-1.8	81.9	25	22+8	13+10	11+29	1	1+7	9m 10s	38.29s
inetutils-1.9.4	117.2	7+4	5+8	9+3	7+10	1	1+5	30.26s	1m 5s
snort-2.9.13	378.2	44+12	33+34	26+4	15+16	1+2	1+1	8m 49s	3m 13s
grub(c6b9a0a)	331.1	13+12	6+5	1	1	0+3	0	3m 27s	1m 1s
Total	1,220.00	166+60	137+124	107+30	85+113	26+9	27+29	31m 12s	10m 44s

Automated repair via deductive synthesis

Algorithm 1 Algorithm for the Deductive Synthesis

Require: $\mathcal{E}, (\pi \wedge \theta_{target})$

Ensure: An expression e_R such that $\mathcal{E} \vdash \{T \wedge \epsilon\}\ e_R\ \{\pi \wedge \theta_{target}\}$

$e_{acc} = ()$

for each $nm\ (x^*) \mapsto [\Phi_{pre}, \Phi_{post}, \Phi_{future}] \in \mathcal{E}$ **do**

  if $\theta_{target} = \epsilon$ **then return** `if` π `then` e_{acc} `else` $()$

  else

    `// there exist a set of program variables` y^*

    $\theta'_{target} = (\pi \wedge [y^*/x^*]\Phi_{post})^{-1}\theta_{target}$

    $e_{acc} = e_{acc};\ nm(y^*)$

  end if

end for

return without any suitable patches

Example: `true` $\wedge$ $\mathcal{E}$ $\not\models$ `ptr≠null` $\wedge$ `_^*.(free(ptr))`

$\Rightarrow$ synthesis(`ptr≠null` $\wedge$ `_^*.(free(ptr))`) $\Rightarrow$ `if (ptr != NULL) free(ptr);`

Automated repair via deductive synthesis

Algorithm 1 Algorithm for the Deductive Synthesis

Require: $\mathcal{E}, (\pi \wedge \theta_{target})$

Ensure: An expression e_R such that $\mathcal{E} \vdash \{T \wedge \epsilon\}\ e_R\ \{\pi \wedge \theta_{target}\}$

```
e_acc = ()
for each nm (x*) ↦ [Φ_pre, Φ_post, Φ_future] ∈ ℰ do
    if θ_target = ϵ then return if π then e_acc else ()
    else
        // there exist a set of program variables y*
        θ'_target = (π ∧ [y*/x*]Φ_post)^-1 θ_target
        e_acc = e_acc; nm(y*)
    end if
end for
return without any suitable patches
```

- Only supporting inserting/deleting calls.
- Do need re-analysis.

Example: `true` $\wedge$ $\mathcal{E}$ $\not\models$ `ptr≠null` $\wedge$ `_^*.(free(ptr))`

$\Rightarrow$ synthesis(`ptr≠null` $\wedge$ `_^*.(free(ptr))`) $\Rightarrow$ `if (ptr != NULL) free(ptr);`

Experiment 2: Repairing bugs

Project	NPD		ML		RL		Time	Infer-v0.9.3			
	#	ProveNFix	#	ProveNFix	#	ProveNFix		#ML	SAVER	#RL	FootPatch
Swoole	53	53	32	28	19	19	4.33s	15+3	11	6+1	6
lxc	26	24	23	22	10	10	3.882s	3+5	3	2+1	0
WavPack	44	41	12	12	0	0	11.435s	1+2	0	2	1
flex	18	18	4	4	1	1	39.38s	3+4	0	0	0
p11-kit	5	4	28	27	6	6	2.452s	33+9	24	2	1
x264	0	0	17	14	5	5	6.375s	10	10	0	0
recutils-1.8	33	30	42	36	8	8	1.261s	10+11	8	1	0
inetutils-1.9.4	15	13	19	17	6	6	1.517s	4+5	4	2+1	1
snort-2.9.13	78	67	42	13	2	2	10.57s	16+27	10	0	0
grub	18	11	1	1	0	0	40.626s	0	0	0	0
Total(Fix Rate)	290	261(90%)	220	174 (79%)	57	57 (100%)	2m 2s	95+66	70(73.7%)	15+3	9(60%)

- ❖ 90% fix - null pointer dereferences,
- ❖ 79% fix - memory leaks
- ❖ 100% fix - resource leaks.

SAVER's pre-analysis time:
26.3 seconds for the flex project
39.5 minutes for the snort-2.9.13 project

Experiment 4: usefulness of spec inference

- 2 predefined primitive specs, OpenSSL-3.1.2, 556.3 kLoC,
- 143.11 seconds to generate future-conditions for 128 OpenSSL APIs
- Example: SSL_CTX_new (meth) ; // future : ((ret=0) /\ return (ret))

OpenSSL Applications	kLoC	Issue ID	Target API	Github Status	ProveNFix	Time
keepalive(843ffc80)	59.1	1003	SSL_CTX_new	✓	✓	5.62s
		1004	SSL_new	✓	✓	
thc-ipv6(011376c)	30.9	28	BN_new	✓	✓	3.32s
		29	BN_set_word	✓	✗	
FreeRADIUS(94149dc)	258.9	2309	BIO_new	✓	✓	38.89s
		2310	i2a_ASN1_OBJECT	✓	✓	
trafficserver(5ee6a5f)	34.1	4292	SSL_CTX_new	✓	✓	21.55s
		4293	SSL_new	✓	✓	
		4294	SSL_write	✓	✓	
sslsplit(19a16bd)	18.7	224	SSL_CTX_use_certificate	✓	✓	2.69s
		225	SSL_use_PrivateKey	✓	✓	
proxytunnel(f7831a2)	3.1	36	SSL_connect	✓	✓	0.62s
		37	SSL_new	✓	✓	

Conclusion

Thanks!

- ❖ Compositional static analyzer via temporal properties.
- ❖ Specified 17 APIs; found 515 bugs from 1 million LOC; with a (on average) 90% fix rate.
- ❖ Specification: a novel *future-condition and* Specification inference.
- ❖ The inferred spec can be used to analysis protocol applications, e.g., OpenSSL.

Future Directions

- ❖ Handle loops without unrolling
- ❖ Enhance expressiveness, e.g., separation logic

Common questions during Q&A

- Why table 1 does not show ProveNFix's false positives?

False positives occur in ProveNFix when there are aliasing/re-assignment, and we take extra care of it via a "CONSUME" event, which entails all other event. The idea is to abandon the proof obligations when there are possible false positives.

- How do you deal with loops?

Loops are unrolled once in this work

- How do you deal with global variables?

As ProveNFix is designed for modular reasoning, without capturing the global states, it assumes global variables are well-managed, i.e., the traces of using them satisfy all the possible constraints.

Common questions during Q&A

- Why It is called bi-abduction, the examples are only to propagate future conditions.

The bi-abduction in theory can infer pre/post-conditions as well, however, this paper primarily focuses on the proof obligations by future conditions.

- Given the rust lifetime reasoning for memory safety, is this work useful in Rust?

Yes, our framework uses general purpose temporal logic, and memory leak is only one case study of the application. In general, if the properties of interest can be encoded using temporal logic, it can be adapted in this framework.

- How to get the initial specifications for the primitives?

We obtained the speculations from the API documentations, which is in natural languages, and we manually convert them into our core syntax. Whether we can automatically generate these specifications is an open question.

Common questions during Q&A

- Does the long call stack happen often? Will the postcondition very long?

- Why ProveNFix can find so much more two bugs?

The criteria of how Infer decided to report a bug is not clear. But one reason could be that Infer only reports errors when the precondition is true, which means that if it is a conditional bug, it chooses to not to report.

Next step

- Formalize future condition, define its bi-abduction on pre/post and future ,
- Achieve repair without re-verification