Compiler-instrumented, Dynamic Secret-Redaction of Legacy Processes for Attacker Deception

Frederico Araujo and Kevin Hamlen
The University of Texas at Dallas

24th USENIX Security Symposium – Washington, D.C., August 12th, 2015 – ONR N00014-14-1-0030, AFOSR FA9550-14-1-0173, NSF 1054629

Problem: Live Process Image Secret-Redaction

- Goal:
  - Remove or replace secrets in address spaces of *running programs*, yielding processes that **CONTINUE RUNNING** (but with no secrets)

- Potential Applications:
  - Debugging: Safely disclose redacted crash dumps to developers
  - Intrusion Response: Dynamic secret redaction without loss of service
  - Cyber Deception: Runtime replacement of secrets with honey-data
    - *Honey-patching [CCS'14]*

Problem: Live Process Image Secret-Redaction

- Goal:
  - Remove or replace secrets in address spaces of *running programs*, yielding processes that **CONTINUE RUNNING** (but with no secrets)

- Potential Applications:
  - Debugging: Safely disclose redacted crash dumps to developers
  - Intrusion Response: Dynamic secret redaction without loss of service
  - **Cyber Deception: Runtime replacement of secrets with honey-data**
    - ***Honey-patching [CCS'14]***

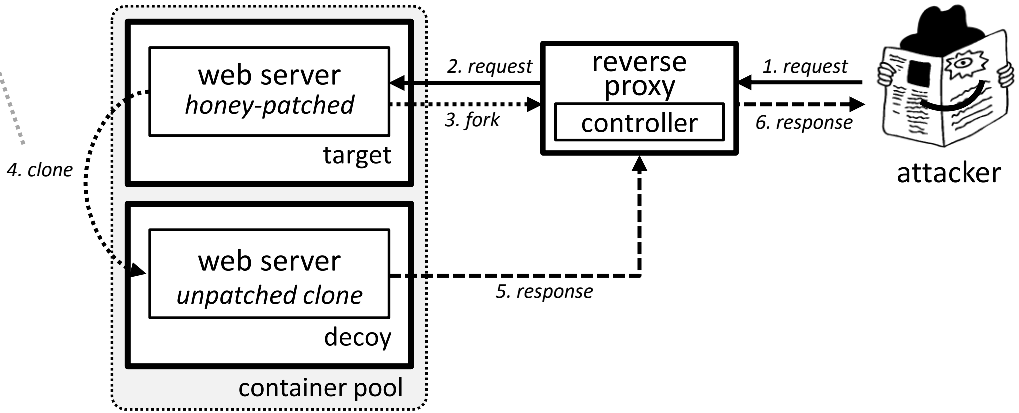
Runtime Secret Redaction

```
..................................L......
.@.. GET / HTTP/1.1 /browse/doc1.html
en_US xyz-198 8229788/6160/11/.......
.Accept-Encoding: gzip,deflate,sdch...
...Accept-Language: en-US,en;q=0.8....
Cookie: app.token= BACC-76GF-ABS3-ZOV2
74f89abc43de7 ,...............*.......
SESSIONID=2321CFA5DA771A284D13DD67798A
557....E.$3Z.l8.M..e5...............
..6D7ED1D554E.........*..?.e.b....L...
..........*......App.token= BACC-65CH-
Accept:text/html,application/xhtml+xml
,application/xml;q=0.9,*/*;q=0.8......
............*.........................
Linux x86_64; rv:32.0) Gecko/20100101
Firefox/32.0...............@.........
.........2d-4f59f9ff30097.......*.....
.GET / HTTP/1.1.......................
```

▨ user 1

▨ user 2

🟥 attacker



Dynamic Taint Tracking

Main idea: *Instrument programs with operations that track (explicit) dataflows of secrets.*

- Program vulnerability detection
  - TaintCheck, LIFT, Mimemu, Argos, …
- Information leak detection
  - TaintDroid, TaintEraser, AndroidLeaks, Spandex, D2Taint, …
- Study of sensitive data lifetime
  - TaintBochs
- Analysis of spyware behavior
  - Panorama, Hookfinder, PHP Aspis, …
- Test set generation
  - Memsherlock, ConfAid, …

Dynamic Taint Tracking

```
/* first colon delimits username:password */
 s1 = memchr(hostinfo, ':', s − hostinfo);
 if (s1) {
     uptr−>user = memdup(hostinfo, s1 − hostinfo);
     ++s1;
     uptr−>password = memdup(s1, s − s1);

 }
```

```
void safe_free(char *s) {
    if (s is a secret)                  // how to test whether s is secret?
        slow_secure_free(s);
    else
        free(s);
}
```

Dynamic Taint Tracking

Taint Introduction:

```
/* first colon delimits username:password */
 s1 = memchr(hostinfo, ':', s − hostinfo);
 if (s1) {
     uptr−>user = memdup(hostinfo, s1 − hostinfo);
     ++s1;
     uptr−>password = memdup(s1, s − s1);
     dfsan_set_label(SECRET, uptr−>password, sizeof(s − s1))
 }
```

```
void safe_free(char *s) {
    if (s is a secret)                    // how to test whether s is secret?
         slow_secure_free(s);
    else
         free(s);
}
```

Dynamic Taint Tracking

Taint Introduction:

```
/* first colon delimits username:password */
 s1 = memchr(hostinfo, ':', s − hostinfo);
 if (s1) {
     uptr−>user = memdup(hostinfo, s1 − hostinfo);
     ++s1;
     uptr−>password = memdup(s1, s − s1);
     dfsan_set_label(SECRET, uptr−>password, sizeof(s − s1))
 }
```

Taint Check:

```
void safe_free(char *s) {
    if (dfsan_get_label(s) == SECRET)
         slow_secure_free(s);
    else
         free(s);
}
```

Retrofitting Headaches

Taint Introduction:

```
…
/* first colon delimits username:password */
 s1 = memchr(hostinfo, ':', s − hostinfo);
 if (s1) {
     uptr−>user = memdup(hostinfo, s1 − hostinfo);
     ++s1;
     uptr−>password = memdup(s1, s − s1);
     dfsan_set_label(SECRET, uptr−>password, sizeof(s − s1))
 }

…
```

Retrofitting Headaches

```
p = &(uptr->password);
*p = malloc(…);
…
/* first colon delimits username:password */
 s1 = memchr(hostinfo, ':', s − hostinfo);
 if (s1) {
     uptr−>user = memdup(hostinfo, s1 − hostinfo);
     ++s1;
     *p = memdup(s1, s − s1);
     dfsan_set_label(SECRET, *p, sizeof(s − s1))
 }

…
```

Taint Introduction:

Apache is 2.2M SLOC!

Our Solution: Declarative Secret Annotations for C

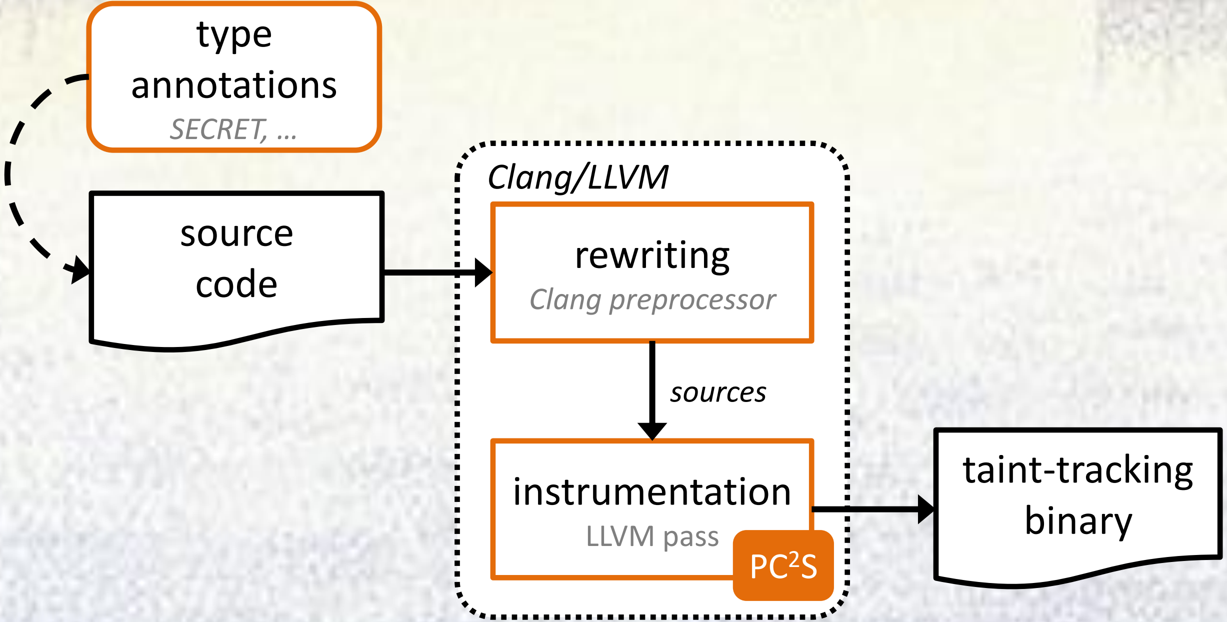
type qualifiers!

```
struct apr_uri_t {
   NONSECRET char *user;
   SECRET_STR char *password;
   …
} SECRET;
```

- *Declarative* vs. *Operational* Secret Annotations
  - Fewer declarations than operations for user to annotate
  - Compiler infers and implements operations from declarations
  - Compiler optimizes operational implementation

- **SECRET** = struct contains secrets
- **SECRET_STR** = field is a pointer to a null-terminated sequence of secret chars
- **NONSECRET** = field is a non-secret within a SECRET struct

SignaC: Secret Information Graph iNstrumentation for Annotated C



Outline

1. Overview
2. Challenges & Approach
3. Implementation
4. Evaluation
5. Application Study
6. Conclusions

Taint Propagation

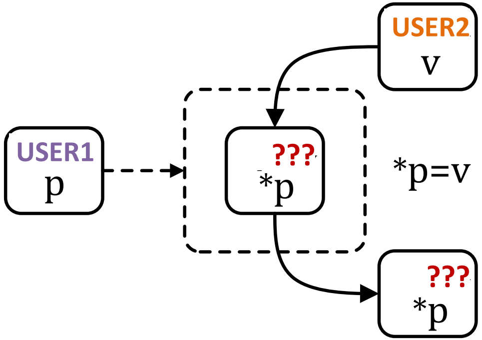
```
p = &(uptr->password);
 *p = malloc(…);


…


/* first colon delimits username:password */
 s1 = memchr(hostinfo, ':', s − hostinfo);
 if (s1) {
      uptr−>user = memdup(hostinfo, s1 − hostinfo);
      ++s1;
      memcpy(*p, s1, s − s1);
}


…
```

Need taint propagation semantics for…

- field access operator (->)
- address-of (&) operator
- assignments (=)
- dereferencing assignments (*p = …)
- dynamic memory allocations (malloc)

Pointer Combine Semantics (PCS)

```
mytype *p = …;
mytype v = …;
dfsan_set_label(USER1, p, sizeof(p));
dfsan_set_label(USER2, v, sizeof(v));
*p = v;
```

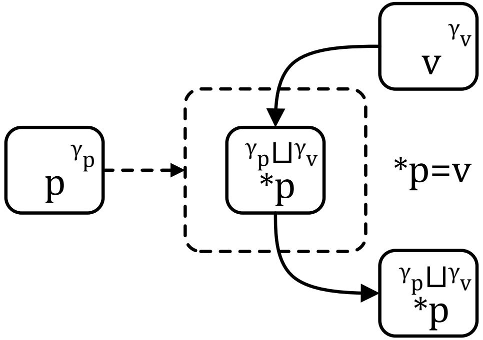


What should be the resulting label of *p ?

- Two standard answers:
  - *No-Combine Semantics:* label of *p is just **USER2**
    - Rationale: **v** was copied; its ownership didn't change.
  - *Combine Semantics:* label of *p is **USER1** ⊔ **USER2** (joint ownership)
    - Rationale: Failing to redact value at *p now possibly divulges value of pointer p.
    - Conclusion: Value *p is now one of **USER1**'s secrets (as well as continuing to be one of **USER2**'s secrets).

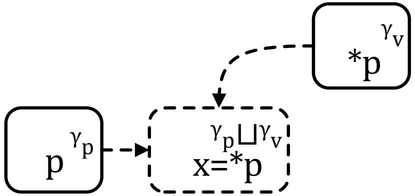
Pointer Combine Semantics (PCS)

propagation semantics:



*p=v

p = &(uptr->password);
*p = malloc(...);
*p[i] = v *(for all i)*



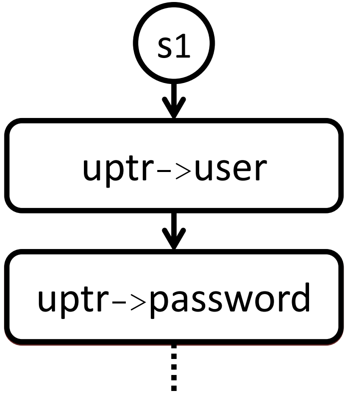
x = *p[i] *(for all i)*

revisiting our initial example…

```
struct apr_uri_t {
  NONSECRET char *user;
  SECRET_STR char *password;
  …
} SECRET;
```

uptr−>user = v1;

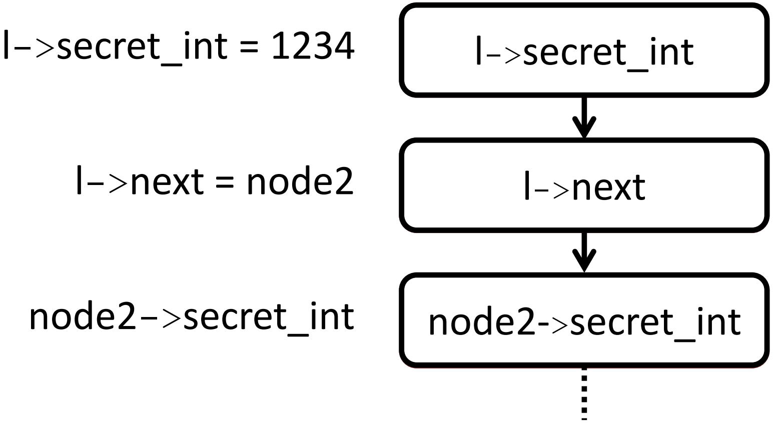
uptr->password = v2;



PCS Problems: Overtainting & Label Creep

but…

```
l->secret_int = 1234;
l->next = node2;
node2->secret_int;
```



l->secret_int = 1234

l->next = node2

node2->secret_int

☹

very common…

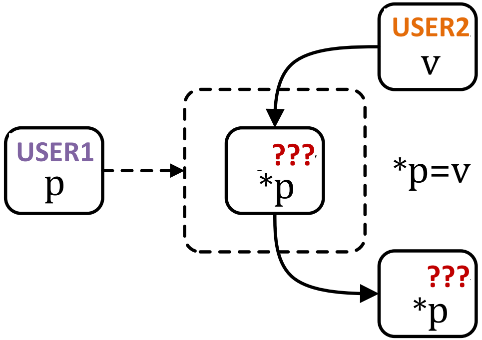
```
while (freelist != NULL) {
   node = freelist;
   freelist = node->next;
   free(node);
}
```

```
while (prev) {
  prev->eos_sent = 1;
  prev = prev->prev;
}
```

…

Our Solution: Pointer Conditional-Combine Semantics (PC^{2}S)

```
mytype *p = …;
mytype v = …;
dfsan_set_label(USER1, p, sizeof(p));
dfsan_set_label(USER2, v, sizeof(v));
*p = v;
```



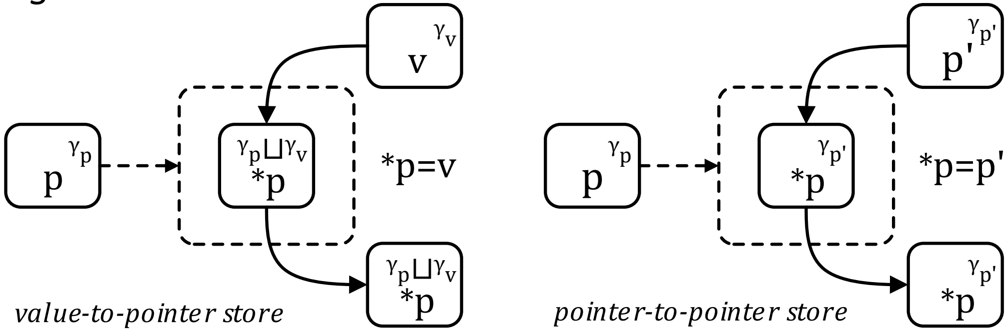
What should be the resulting label of ***p** ?

A New Third Answer:

- *Conditional-Combine Semantics:* label of ***p** depends upon the *static type* of **v**!
  - if **v** has *pointer* type, then use *No-Combine Semantics* (**USER2**).
  - if **v** has *non-pointer* type, then use *Combine Semantics* (**USER1** ⊔ **USER2**).

Our Solution: Pointer Conditional-Combine Semantics (PC^{2}S)

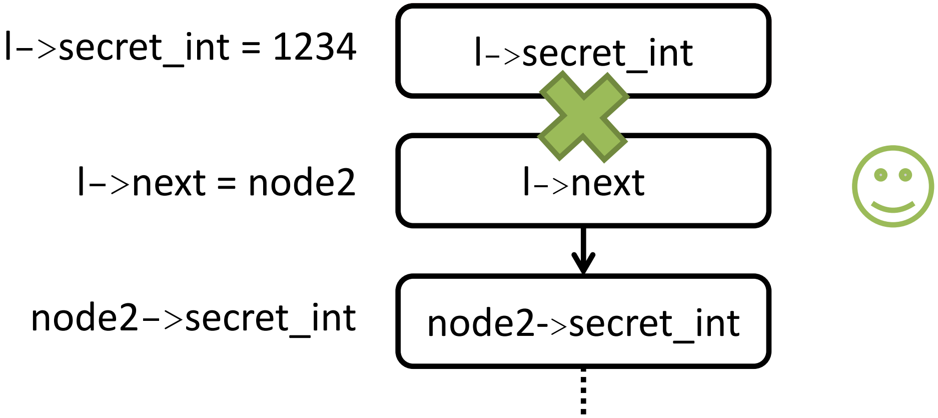
propagation semantics:



taint policy: "do not combine when pointee has pointer type"

let's try again…

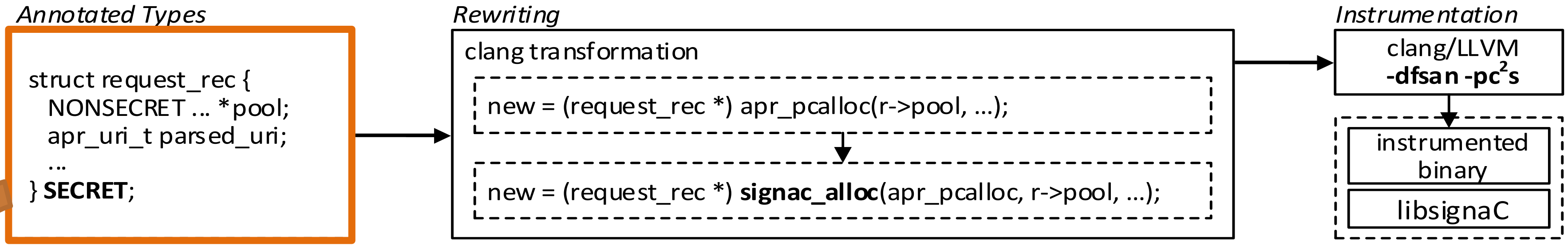
```
l->secret_int = 1234;
l->next = node2;
node2->secret_int;
```



Outline

1. Overview
2. Challenges & Approach
3. Implementation
4. Evaluation
5. Application Study
6. Conclusions

SignaC: Type Attributes



```
struct request_rec {
  NONSECRET ... *pool;
  apr_uri_t parsed_uri;
  ...
} SECRET;
```

```
new = (request_rec *) apr_pcalloc(r->pool, ...);
```

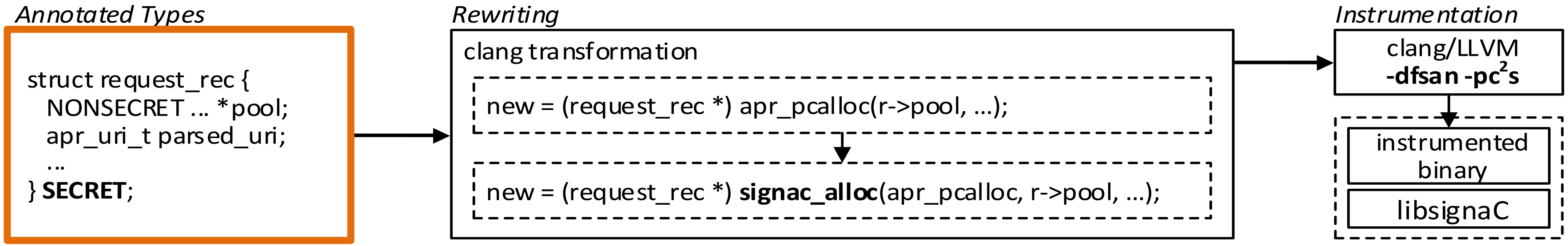
```
new = (request_rec *) signac_alloc(apr_pcalloc, r->pool, ...);
```

#define **SECRET** **__attribute__((annotate("secret")))**

clang -Xclang -ast-dump

```
|-RecordDecl 0x8943a40 <line:15:9, line:19:1> line:15:16 struct request_rec definition
| |-AnnotateAttr 0x8943d60 <line:3:31, col:48> "secret"
| |-FieldDecl 0x8943c20 <line:16:5, col:29> col:29 referenced pool 'apr_pool_t * ...
| `-FieldDecl 0x8943ca0 <line:17:5, col:15> col:15 parsed_uri 'apr_uri_t':'struct apr_uri_t'
```

SignaC: Type Attributes



Annotated Types

```
struct request_rec {
  NONSECRET ... *pool;
  apr_uri_t parsed_uri;
  ...
} SECRET;
```

Rewriting

clang transformation

```
new = (request_rec *) apr_pcalloc(r->pool, ...);
```

```
new = (request_rec *) signac_alloc(apr_pcalloc, r->pool, ...);
```

Instrumentation

clang/LLVM
-dfsan -pc^{2}s

instrumented binary

libsignaC

```
#define NONSECRET __attribute__((type_annotate("nonsecret")))
#define SECRET_STR __attribute__((type_annotate("secret_str")))
```

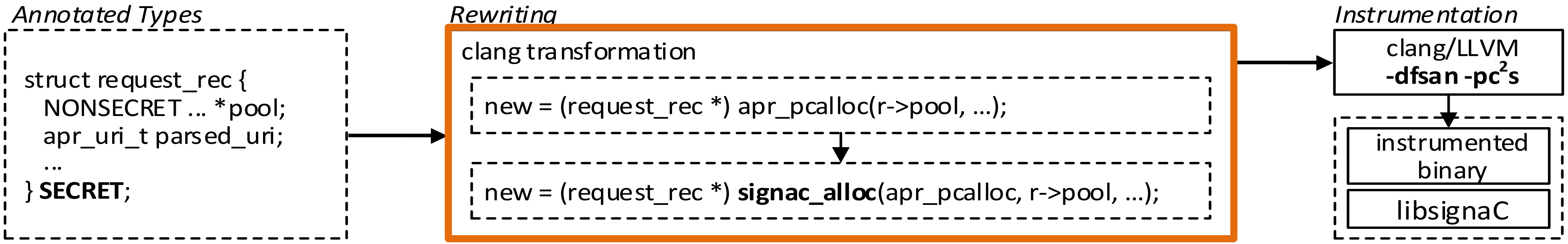
} *Quala type qualifiers*

```
rec->pool = pool;
```

clang -S -emit-llvm

```
%3 = load %struct.apr_pool_t** %pool, align 8
%4 = load %struct.request_rec** %rec, align 8
%pool2 = getelementptr inbounds %struct.request_rec* %4, i32 0, i32 0
store %struct.apr_pool_t* %3, %struct.apr_pool_t** %pool2, align 8, !tyann !1
...
!1 = !{!"nonsecret", i8 0}
```

SignaC: Type Attribute Rewriting



Annotated Types

```
struct request_rec {
  NONSECRET ... *pool;
  apr_uri_t parsed_uri;
  ...
} SECRET;
```

Rewriting

clang transformation

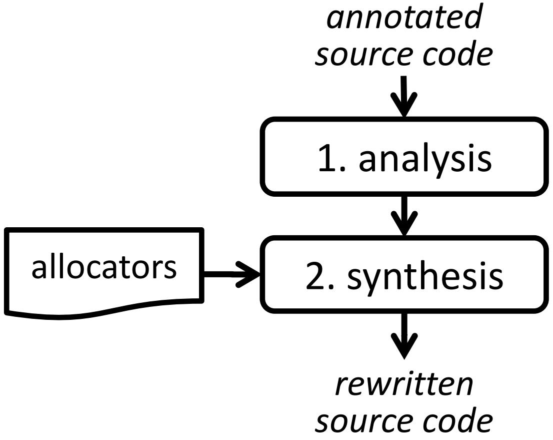
```
new = (request_rec *) apr_pcalloc(r->pool, ...);
```

```
new = (request_rec *) signac_alloc(apr_pcalloc, r->pool, ...);
```

Instrumentation

→ Clang tooling API: AST Matchers + Rewriting API

→ Allocators list: {malloc, calloc, apr_palloc, apr_pcalloc, …}



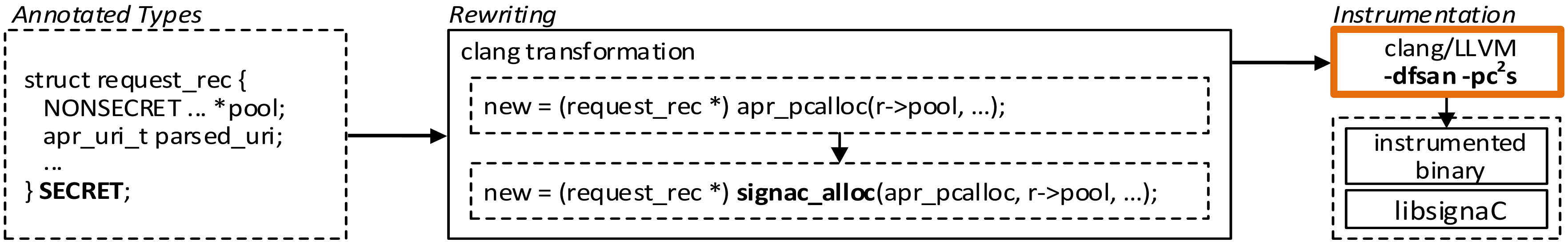
1. Compute the list of all ***security-relevant datatypes***

```
struct  s {      union u {      typedef  s    S;      struct  S' {      union  S' {
  ...              ...          typedef  u    U;        S ns;             S ns;
} SECRET;        } SECRET;                              ... };            ... };
```

2. Wrap all security-relevant datatypes instantiations with **signac_alloc**

```
#define signac_alloc(alloc, args...) ({ \
        void * __p = alloc ( args ); \
        signac_taint(&__p, sizeof(void*)); \
        __p; })
```

SignaC: PC²S Instrumentation



Annotated Types

```
struct request_rec {
  NONSECRET … *pool;
  apr_uri_t parsed_uri;
  …
} SECRET;
```

Rewriting

clang transformation

```
new = (request_rec *) apr_pcalloc(r->pool, …);
```

```
new = (request_rec *) signac_alloc(apr_pcalloc, r->pool, …);
```

Instrumentation

clang/LLVM
-dfsan -pc²s

instrumented binary

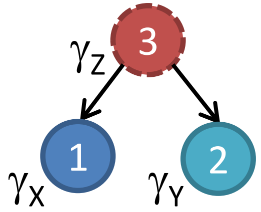
libsignaC

- → implemented as an extension to **DFSan**
- → low-overhead representation of labels: **16-bit integers** allocated sequentially
- → maps without reserving the lower 32TB of the process address space for ***shadow memory***
- → *union labels* organized as a dynamically growing binary (DAG) – the ***union table***

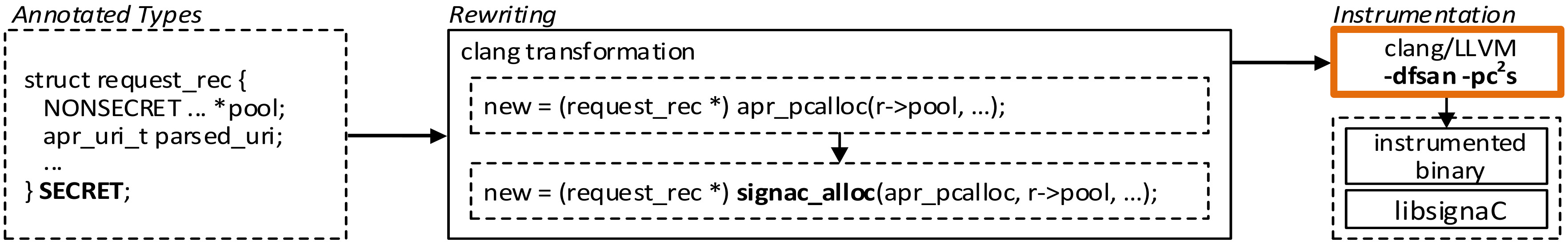
Start	End	Memory Region
0x700000008000	0x800000000000	application memory
0x200000000000	0x700000008000	union table
0x000000010000	0x200000000000	shadow memory
0x000000000000	0x000000010000	reserved by kernel

example:

$Z = X + Y$



SignaC: PC^{2}S Instrumentation



Annotated Types

```
struct request_rec {
  NONSECRET … *pool;
  apr_uri_t parsed_uri;
  …
} SECRET;
```

Rewriting

clang transformation

```
new = (request_rec *) apr_pcalloc(r->pool, …);
```

```
new = (request_rec *) signac_alloc(apr_pcalloc, r->pool, …);
```

Instrumentation

clang/LLVM
-dfsan -pc^{2}s

instrumented binary

libsignaC

→ label propagation across external library interfaces expressed as an **ABI list**

→ DFSan predefines an ABI list that covers glibc

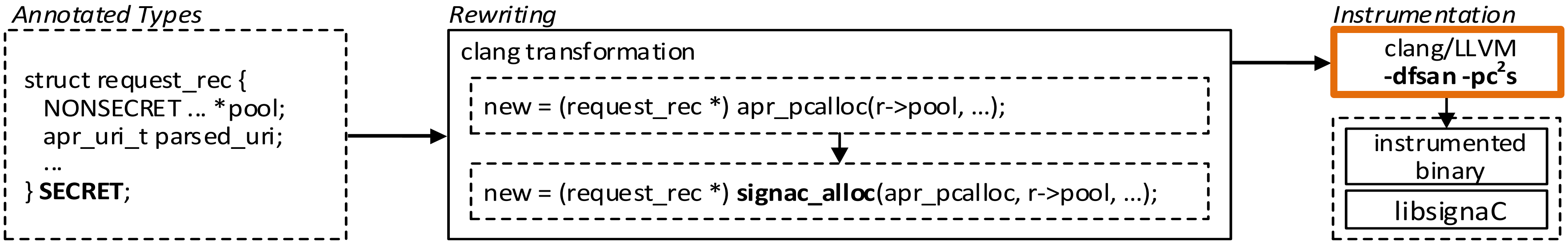
```
fun:malloc=custom
fun:realloc=discard
fun:free=discard
…
fun:isalpha=functional
fun:isdigit=functional
…
fun:memcpy=custom
fun:memset=custom
fun:strcpy=custom
```

discard	$\rho_{dis}(\overline{\gamma}) \coloneqq \bot$
functional	$\rho_{fun}(\overline{\gamma}) \coloneqq \bigsqcup \overline{\gamma}$
custom	*custom-defined label propagation wrapper*

→ other libraries mapped to the ABI: OpenSSL, PCRE, APR, …

→ *memory transfer functions* (e.g., *strcpy*, *strdup*) and *input functions* (e.g., *read*, *pread*) ABI extensions for PC^{2}S

SignaC: PC^{2}S Instrumentation



Annotated Types

```
struct request_rec {
  NONSECRET ... *pool;
  apr_uri_t parsed_uri;
  ...
} SECRET;
```

Rewriting

clang transformation

`new = (request_rec *) apr_pcalloc(r->pool, ...);`

`new = (request_rec *) signac_alloc(apr_pcalloc, r->pool, ...);`

Instrumentation

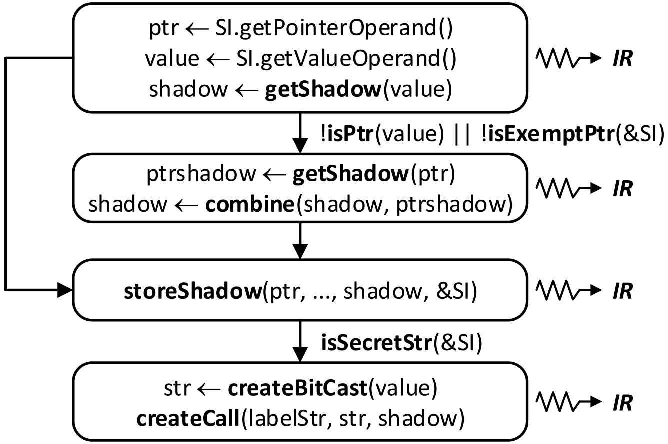
clang/LLVM
-dfsan -pc^{2}s

instrumented binary

libsignaC

- → instrumentation operates on **LLVM IR**, inserting label propagation code
- → propagation policy parametrized at the compiler's front-end: ***pc2s-on-store***, ***pc2s-on-load***

example:
store instruction



ptr ← SI.getPointerOperand()
value ← SI.getValueOperand()
shadow ← **getShadow**(value)

IR

!**isPtr**(value) || !**isExemptPtr**(&SI)

ptrshadow ← **getShadow**(ptr)
shadow ← **combine**(shadow, ptrshadow)

IR

storeShadow(ptr, ..., shadow, &SI)

IR

isSecretStr(&SI)

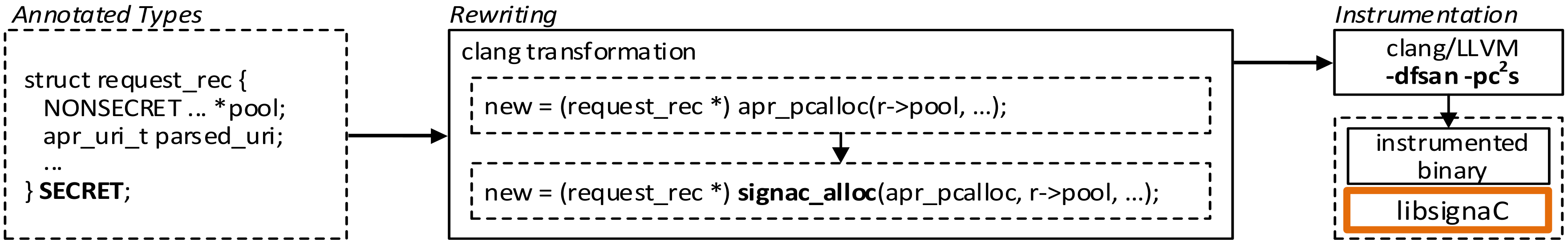
str ← **createBitCast**(value)
createCall(labelStr, str, shadow)

IR

*check for **NON_SECRET** annotated pointers*

*special handling of **SECRET_STR** annotated types*

SignaC: Runtime Library



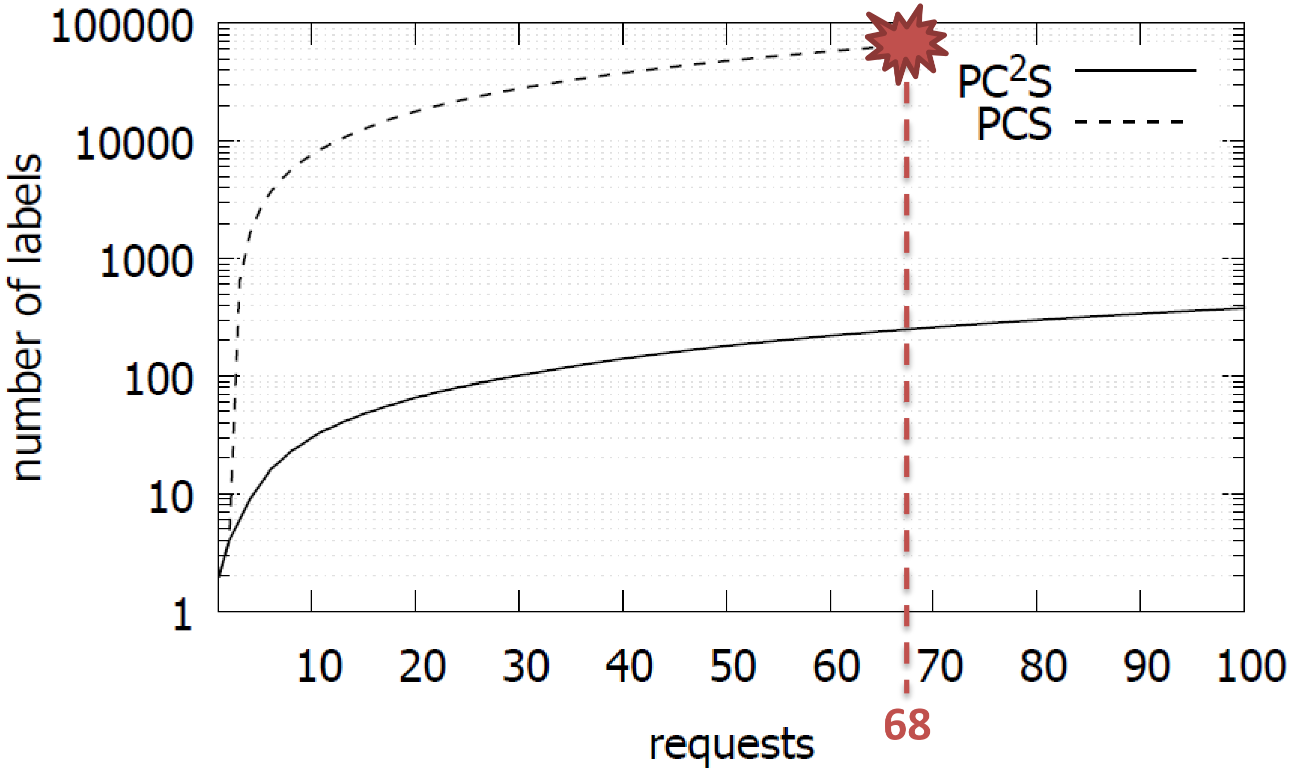
→ libsignaC : tiny C library that encapsulates runtime support for the type annotation mechanism

signac_init(*pl*)	initialize a tainting context with a fresh label instantiation *pl* for the current principal.
signac_taint(*addr*, *size*)	taint each address in interval [*addr*; *addr*+*size*) with pl.
signac_alloc(*alloc*, ...)	wrap allocator alloc and taint the address of its returned pointer with *pl*.

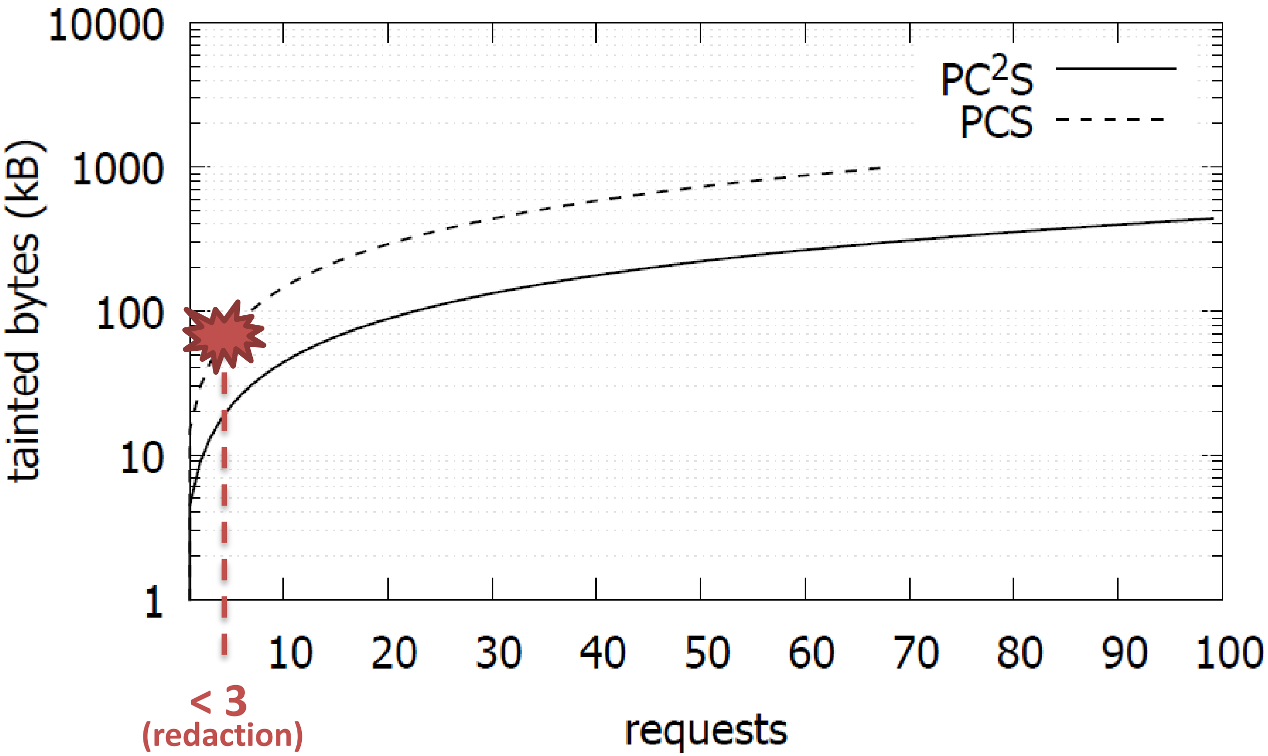
Outline

1. Overview
2. Challenges & Approach
3. Implementation
4. Evaluation
5. Application Study
6. Conclusions

Label Creep


100000
10000
1000
100
10
1
number of labels
10
20
30
40
50
60
70
80
90
100
requests
PC²S
PCS
68

Taint Spread



Instrumentation Overhead

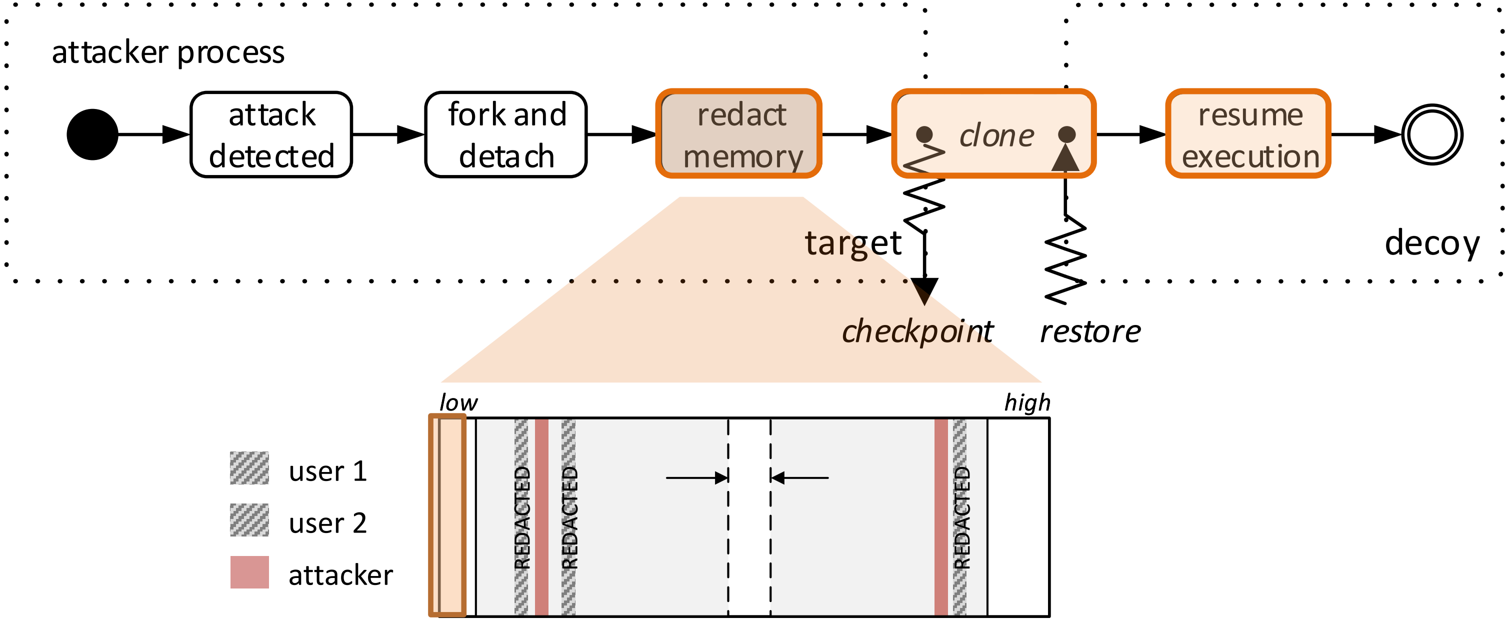
Table 2: Average overhead of instrumentation

Benchmark	***C*=1**	***C=10***	***C*=50**	***C*=100**
Static	2.50	2.34	2.56	2.32
CGI Bash	1.29	0.98	1.00	0.97
PHP	0.41	0.37	0.30	0.31

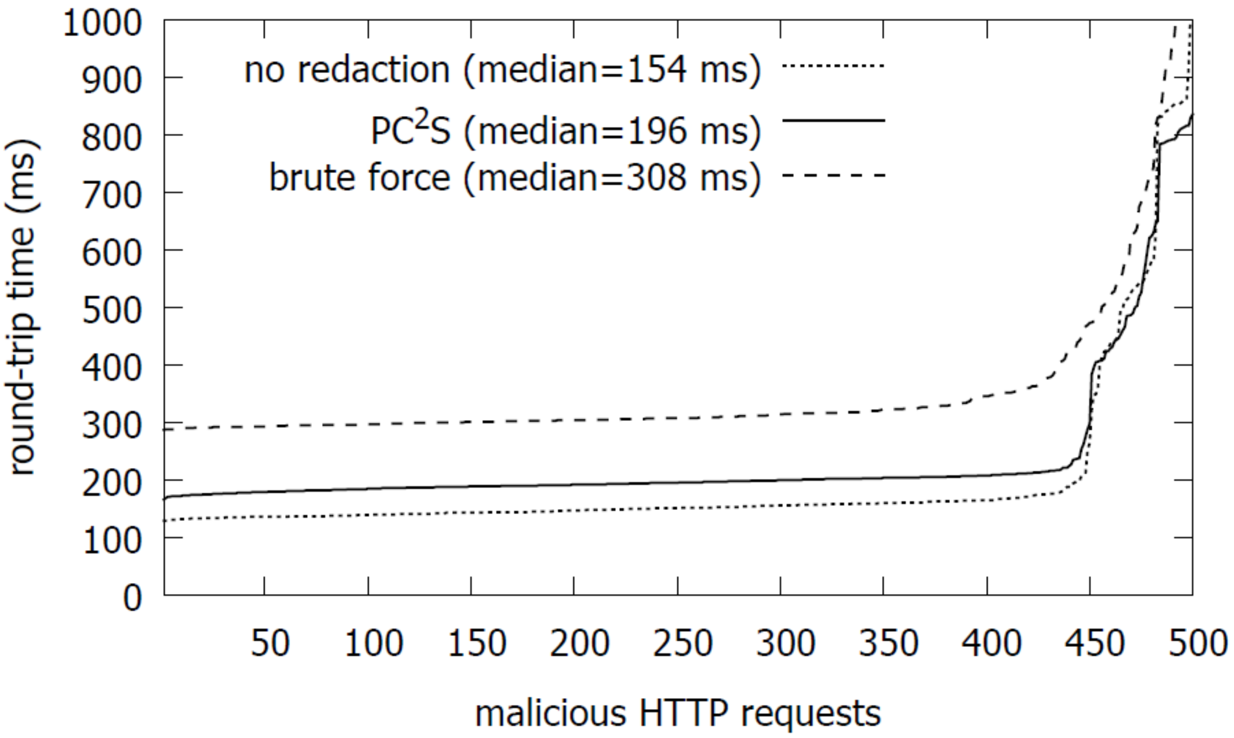
Outline

1. Overview
2. Challenges & Approach
3. Implementation
4. Evaluation
5. Application Study
6. Conclusions

Secret Redaction

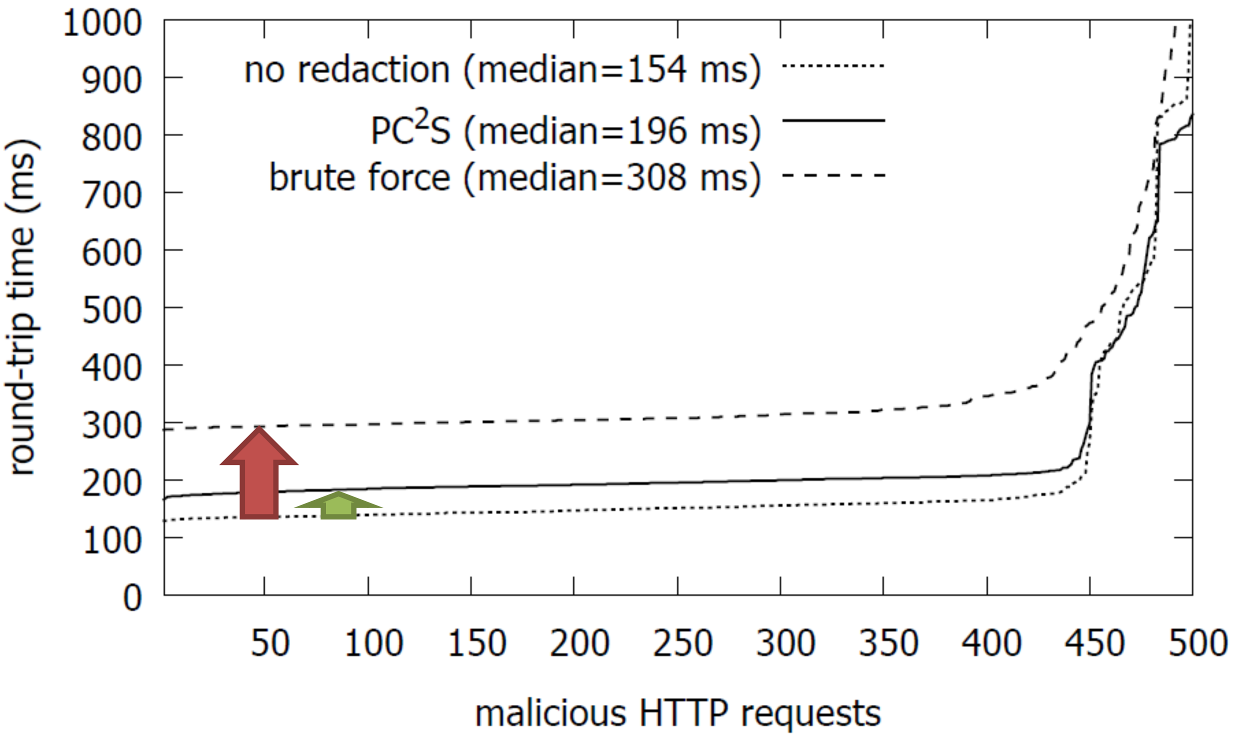


Secret Redaction Performance



Deception and Performance

Deception Strategy: Artificially delay non-forking responses to match the forking delay.



Validation Tests

Table 1: Honey-patched security vulnerabilities

Software	**Version**	**CVE-ID**	**Description**
Apache	2.2.21	CVE-2011-3368	Improper URL Validation
	2.2.9	CVE-2010-2791	Improper timeouts of keep-alive connections
	2.2.15	CVE-2010-1452	Bad request handling
	2.2.11	CVE-2009-1890	Request content length out of bounds
	2.0.55	CVE-2005-3357	Bad SSL protocol check
OpenSSL	1.0.1f	CVE-2014-0160	Buffer over-read in heartbeat extension
Bash	4.3	CVE-2014-6271	Improper parsing of environment variables

Outline

1. Overview
2. Challenges & Approach
3. Implementation
4. Evaluation
5. Application Study
6. Conclusions

Conclusions

- ***Declarative annotation of secrets***
  - New pointer tainting methodology
  - Reduced secret annotation burden
- **New taint propagation semantics**
  - Accurate while containing taint spread and label creep
  - Implemented in LLVM
- Implemented a **memory redactor** for secure honey-patching
- **Tested** on three production web servers

The End

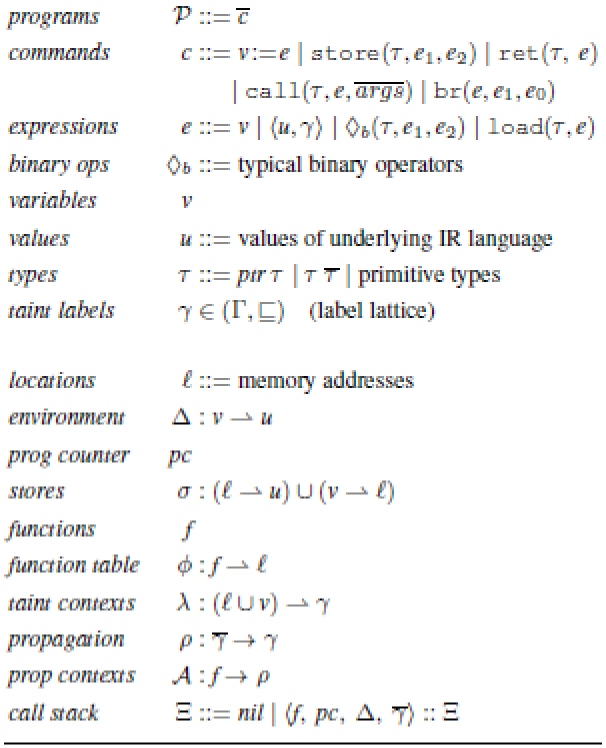
Thank you!

Questions?

Frederico Araujo

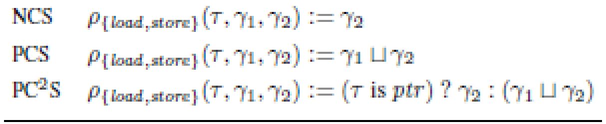
(frederico.araujo@utdallas.edu)

Operational Semantics



programs	$\mathcal{P} ::= \overline{c}$
commands	$c ::= v{:=}e \mid \texttt{store}(\tau, e_1, e_2) \mid \texttt{ret}(\tau,\ e)$ $\mid \texttt{call}(\tau, e, \overline{args}) \mid \texttt{br}(e, e_1, e_0)$
expressions	$e ::= v \mid \langle u, \gamma \rangle \mid \lozenge_b(\tau, e_1, e_2) \mid \texttt{load}(\tau, e)$
binary ops	$\lozenge_b ::=$ typical binary operators
variables	v
values	$u ::=$ values of underlying IR language
types	$\tau ::= ptr\, \tau \mid \tau\, \overline{\tau} \mid$ primitive types
taint labels	$\gamma \in (\Gamma, \sqsubseteq)$ (label lattice)
locations	$\ell ::=$ memory addresses
environment	$\Delta : v \rightharpoonup u$
prog counter	pc
stores	$\sigma : (\ell \rightharpoonup u) \cup (v \rightharpoonup \ell)$
functions	f
function table	$\phi : f \rightharpoonup \ell$
taint contexts	$\lambda : (\ell \cup v) \rightharpoonup \gamma$
propagation	$\rho : \overline{\gamma} \to \gamma$
prop contexts	$\mathcal{A} : f \to \rho$
call stack	$\Xi ::= nil \mid \langle f,\ pc,\ \Delta,\ \overline{\gamma} \rangle :: \Xi$

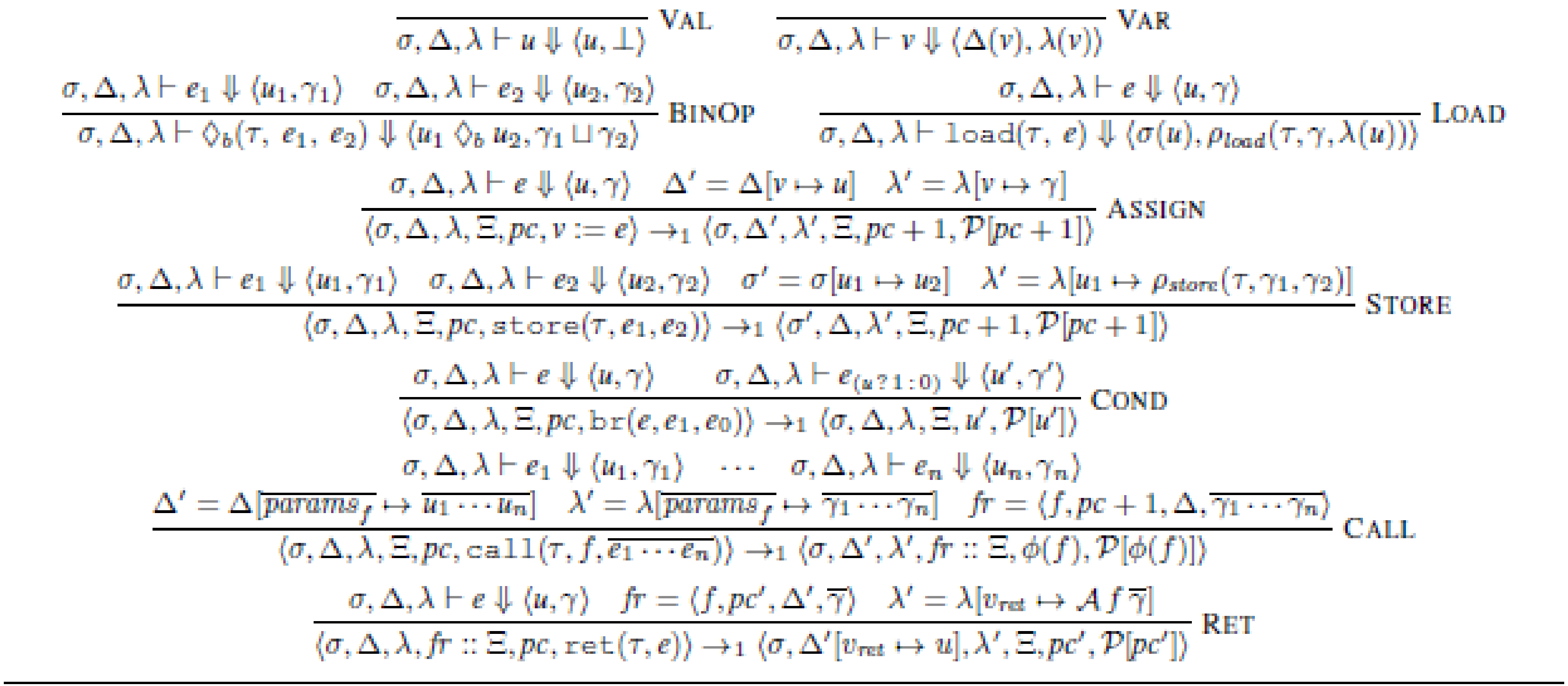
Figure 2: Intermediate representation syntax.



NCS	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := \gamma_2$
PCS	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := \gamma_1 \sqcup \gamma_2$
PC^{2}S	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := (\tau \text{ is } ptr)\ ?\ \gamma_2 : (\gamma_1 \sqcup \gamma_2)$

Figure 4: Polymorphic functions modeling no-combine, pointer-combine, and PC^{2}S label propagation policies.

Operational Semantics



$$\frac{}{\sigma, \Delta, \lambda \vdash u \Downarrow \langle u, \bot \rangle}\ \textsc{Val} \qquad \frac{}{\sigma, \Delta, \lambda \vdash v \Downarrow \langle \Delta(v), \lambda(v) \rangle}\ \textsc{Var}$$

$$\frac{\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle \quad \sigma, \Delta, \lambda \vdash e_2 \Downarrow \langle u_2, \gamma_2 \rangle}{\sigma, \Delta, \lambda \vdash \Diamond_b(\tau,\ e_1,\ e_2) \Downarrow \langle u_1 \Diamond_b u_2, \gamma_1 \sqcup \gamma_2 \rangle}\ \textsc{BinOp} \qquad \frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle}{\sigma, \Delta, \lambda \vdash \texttt{load}(\tau,\ e) \Downarrow \langle \sigma(u), \rho_{load}(\tau, \gamma, \lambda(u)) \rangle}\ \textsc{Load}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \quad \Delta' = \Delta[v \mapsto u] \quad \lambda' = \lambda[v \mapsto \gamma]}{\langle \sigma, \Delta, \lambda, \Xi, pc, v := e \rangle \rightarrow_1 \langle \sigma, \Delta', \lambda', \Xi, pc + 1, \mathcal{P}[pc + 1] \rangle}\ \textsc{Assign}$$

$$\frac{\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle \quad \sigma, \Delta, \lambda \vdash e_2 \Downarrow \langle u_2, \gamma_2 \rangle \quad \sigma' = \sigma[u_1 \mapsto u_2] \quad \lambda' = \lambda[u_1 \mapsto \rho_{store}(\tau, \gamma_1, \gamma_2)]}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{store}(\tau, e_1, e_2) \rangle \rightarrow_1 \langle \sigma', \Delta, \lambda', \Xi, pc + 1, \mathcal{P}[pc + 1] \rangle}\ \textsc{Store}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \qquad \sigma, \Delta, \lambda \vdash e_{(u\,?\,1\,:\,0)} \Downarrow \langle u', \gamma' \rangle}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{br}(e, e_1, e_0) \rangle \rightarrow_1 \langle \sigma, \Delta, \lambda, \Xi, u', \mathcal{P}[u'] \rangle}\ \textsc{Cond}$$

$$\frac{\begin{array}{c}\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle \quad \cdots \quad \sigma, \Delta, \lambda \vdash e_n \Downarrow \langle u_n, \gamma_n \rangle \\ \Delta' = \Delta[\overline{params_f} \mapsto \overline{u_1 \cdots u_n}] \quad \lambda' = \lambda[\overline{params_f} \mapsto \overline{\gamma_1 \cdots \gamma_n}] \quad fr = \langle f, pc + 1, \Delta, \overline{\gamma_1 \cdots \gamma_n} \rangle\end{array}}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{call}(\tau, f, \overline{e_1 \cdots e_n}) \rangle \rightarrow_1 \langle \sigma, \Delta', \lambda', fr :: \Xi, \phi(f), \mathcal{P}[\phi(f)] \rangle}\ \textsc{Call}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \quad fr = \langle f, pc', \Delta', \overline{\gamma} \rangle \quad \lambda' = \lambda[v_{ret} \mapsto \mathcal{A}\, f\, \overline{\gamma}]}{\langle \sigma, \Delta, \lambda, fr :: \Xi, pc, \texttt{ret}(\tau, e) \rangle \rightarrow_1 \langle \sigma, \Delta'[v_{ret} \mapsto u], \lambda', \Xi, pc', \mathcal{P}[pc'] \rangle}\ \textsc{Ret}$$

Figure 3: Operational semantics of a generalized label propagation semantics.