ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code

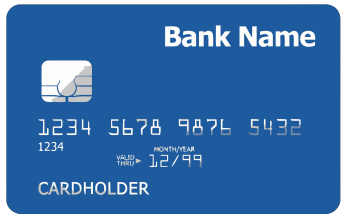
Ajay Brahmakshatriya[1*], Piyus Kedia[2*], Derrick McKee[4*], Deepak Garg[5], Akash Lal[6], Aseem Rastogi[6], Hamed Nemati[3], Anmol Panda[6], Pratik Bhatu[7*]

EuroSys-2019

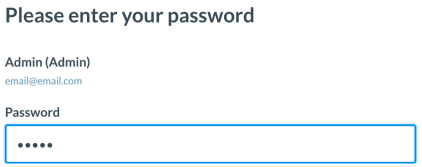
MIT, USA[1] | IIIT Delhi, India[2] | CISPA Saarland University, Germany[3] | Purdue University, USA[4] | MPI-SWS, Germany[5] | Microsoft Research, India[6] | AMD, India[7]
* Work done while the author was at Microsoft Research, India.

Computing on Confidential Information

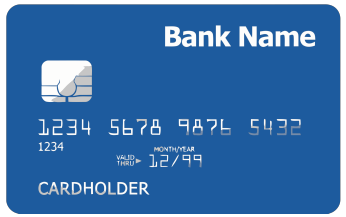
Computing on Confidential Information

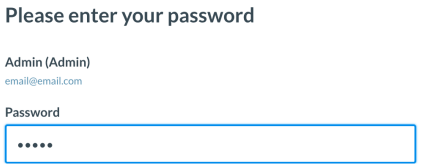


User credentials

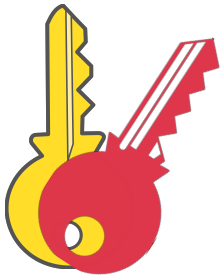


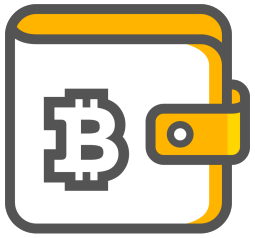
Computing on Confidential Information





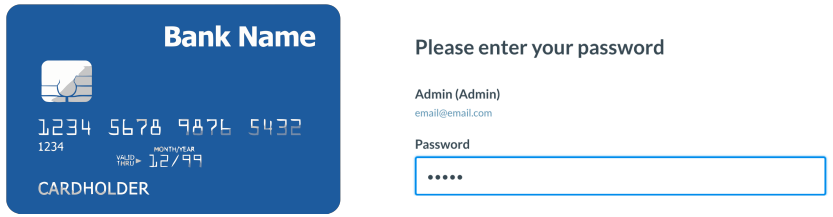
User credentials



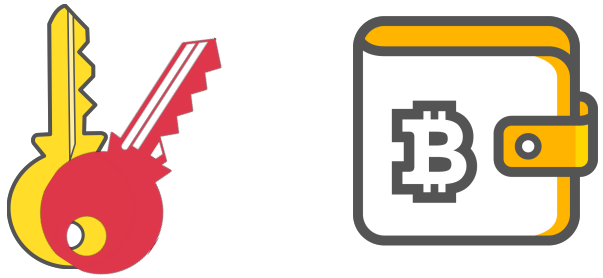


Cryptographic key pairs

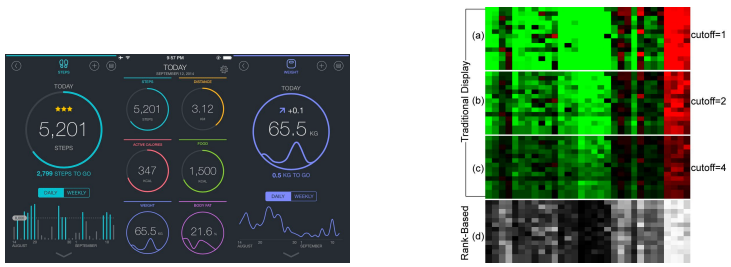
Computing on Confidential Information



User credentials



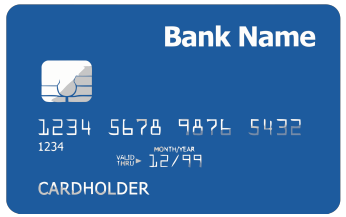
Cryptographic key pairs

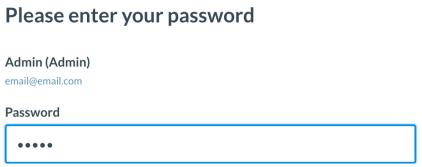


Confidential user health and genomic data

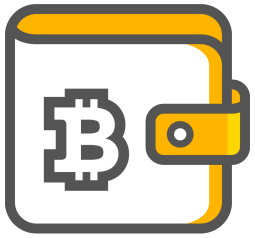
Image from - https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-6-115

Computing on Confidential Information

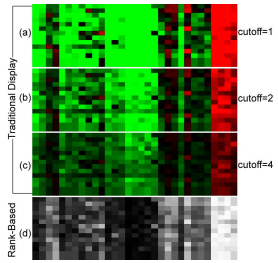




User credentials



Cryptographic key pairs



Confidential user health and genomic data

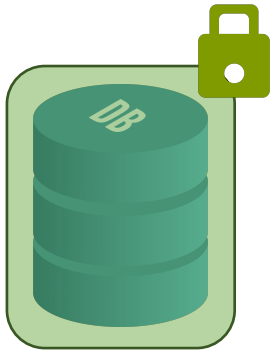
Cloud infrastructure

Image from - https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-6-115

Securing the data – *Always*!

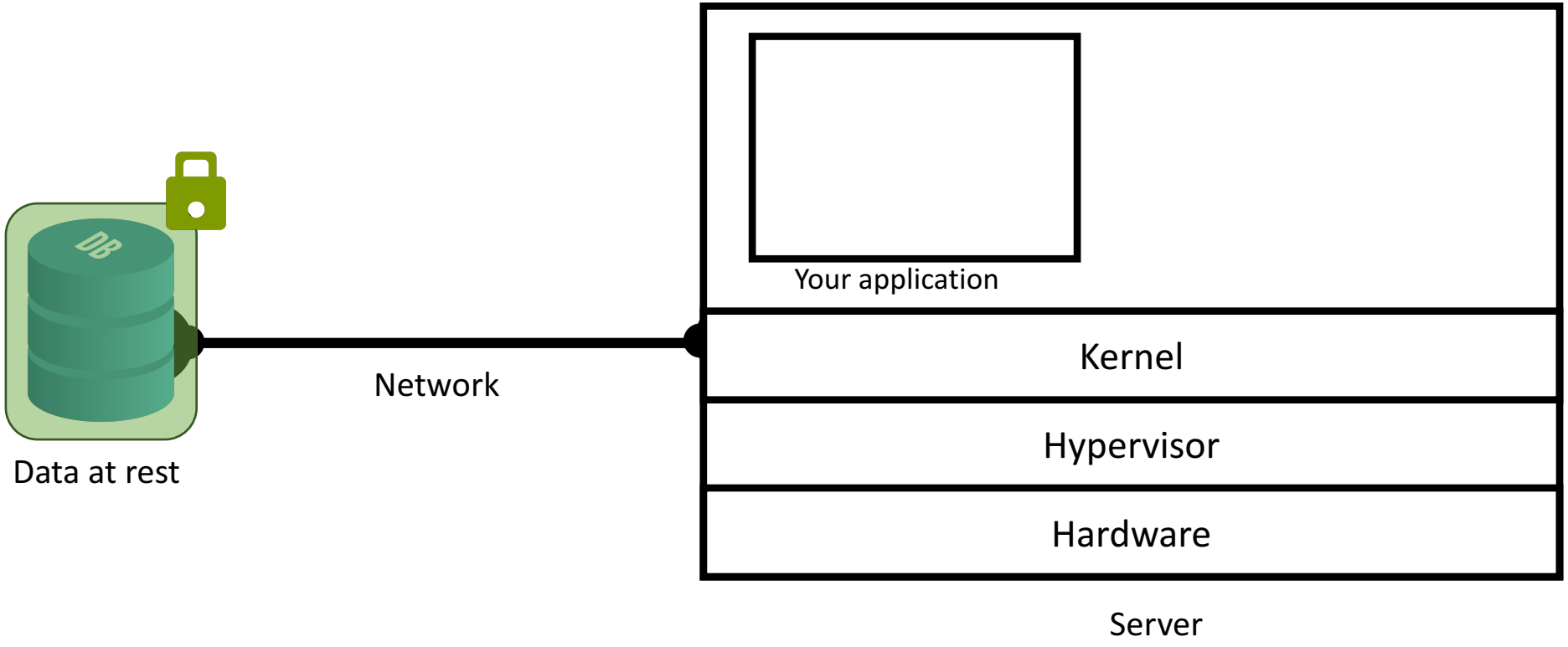
Data at rest

Securing the data – *Always*!

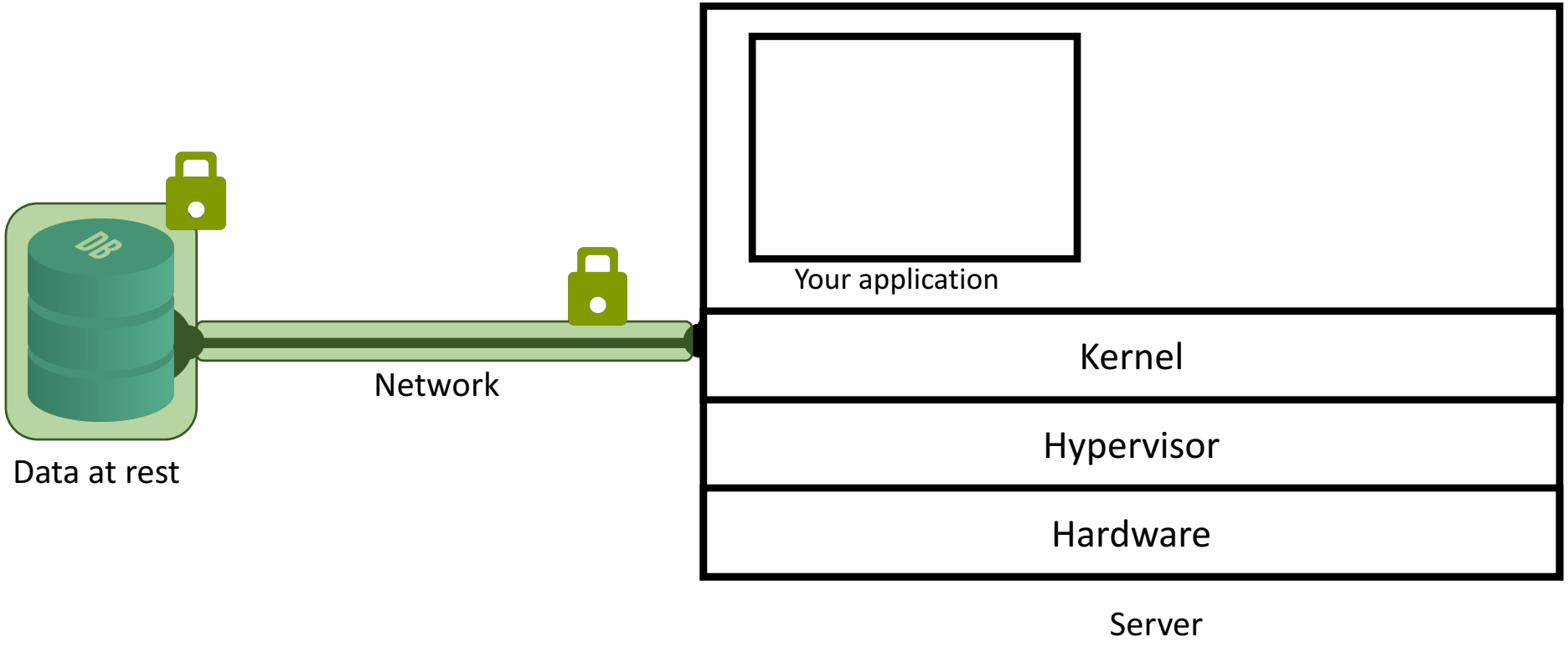


Data at rest

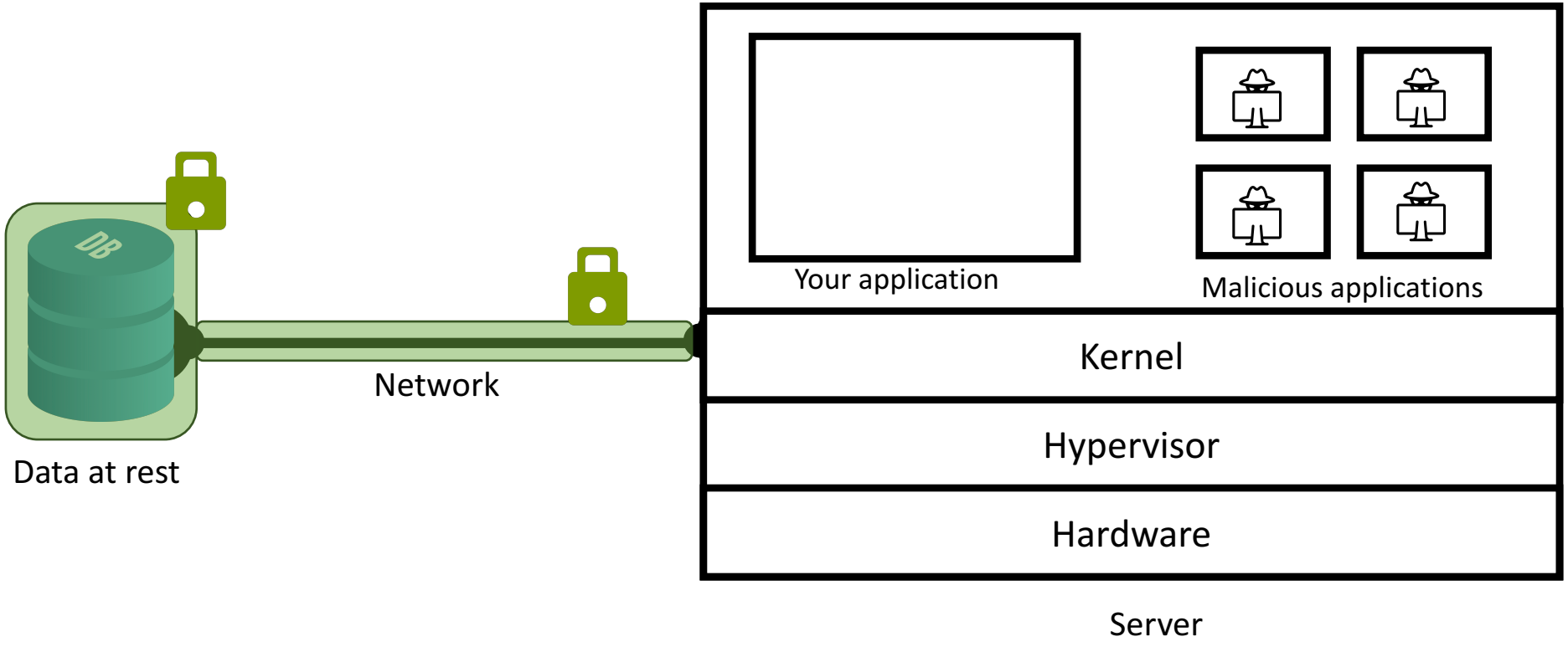
Securing the data – *Always*!



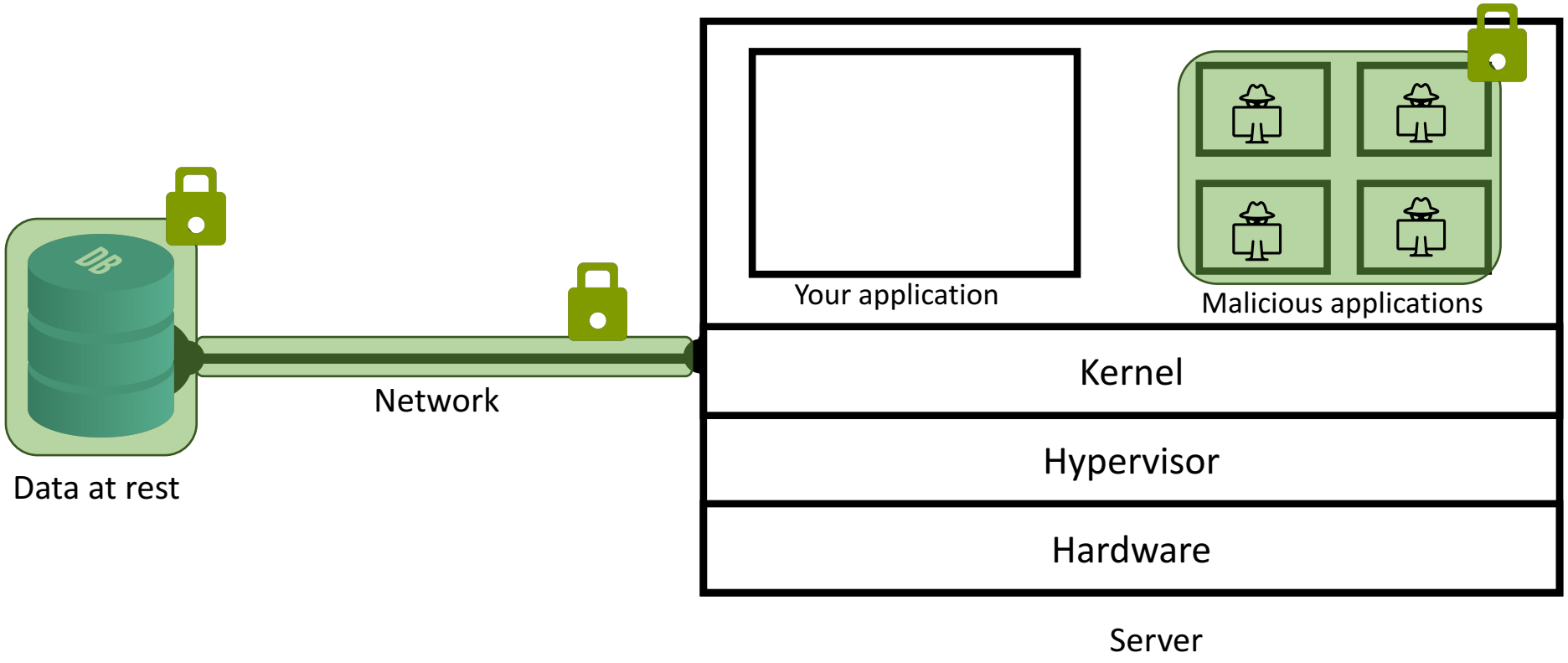
Securing the data – *Always*!



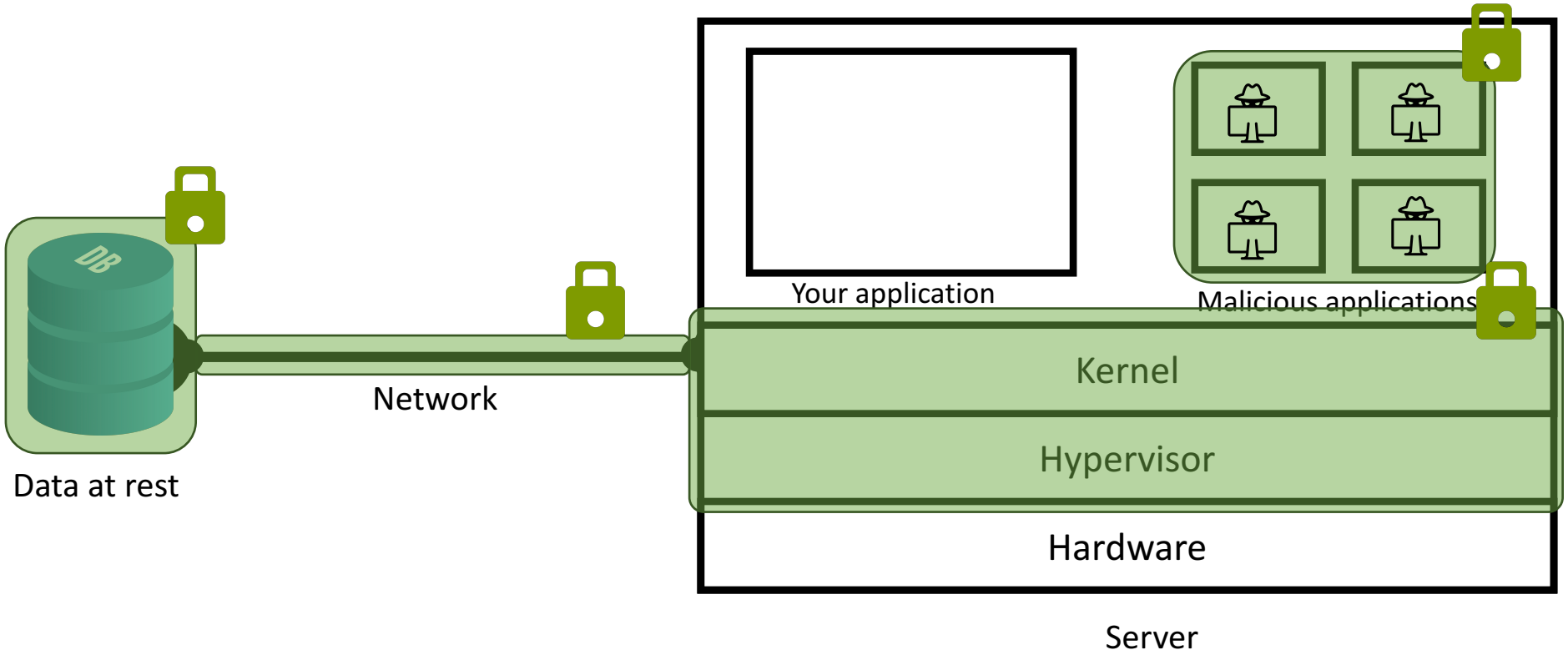
Securing the data – *Always*!



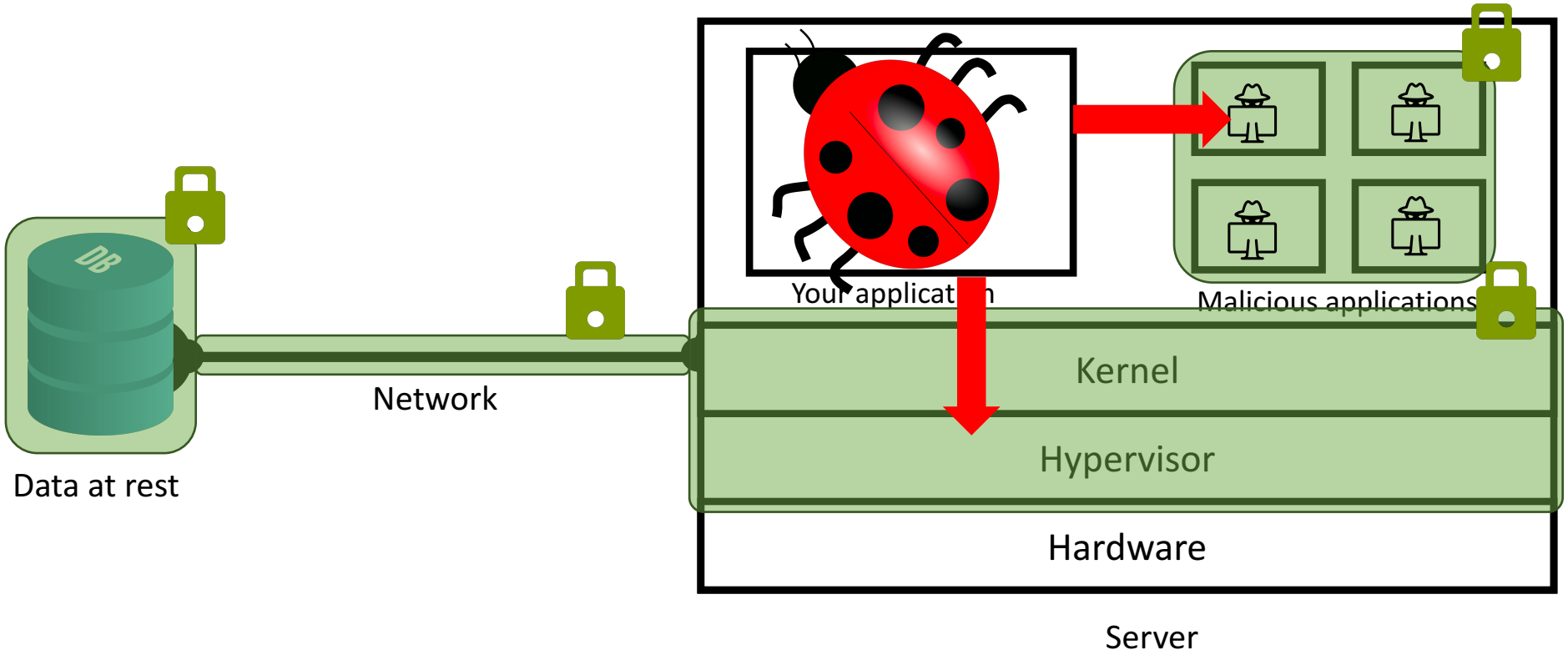
Securing the data – *Always*!



Securing the data – *Always*!



Securing the data – *Always*!



Exploiting bugs in applications

```
void handleReq(char *uname, char * upasswd, char *fname, char *out, int out_size) {
    char passwd [SIZE] , fcontents [SIZE];
    read_password ( uname , passwd , SIZE );
    if (!( authenticate (uname, upasswd, passwd ))) {
        return;
    }

    send(log_file, passwd, SIZE );

    read_file(fname, fcontents, SIZE );


    memcpy(out, fcontents, out_size );


    sprintf(out + SIZE, fmt, "Request complete");
    return;
}
```

Exploiting bugs in applications

```
void handleReq(char *uname, char * upasswd, char *fname, char *out, int out_size) {
    char passwd [SIZE] , fcontents [SIZE];
    read_password ( uname , passwd , SIZE );
    if (!( authenticate (uname, upasswd, passwd ))) {
        return;
    }
    // inadvertently copying the password to the log file
    send(log_file, passwd, SIZE );

    read_file(fname, fcontents, SIZE );


    memcpy(out, fcontents, out_size );


    sprintf(out + SIZE, fmt, "Request complete");
    return;
}
```

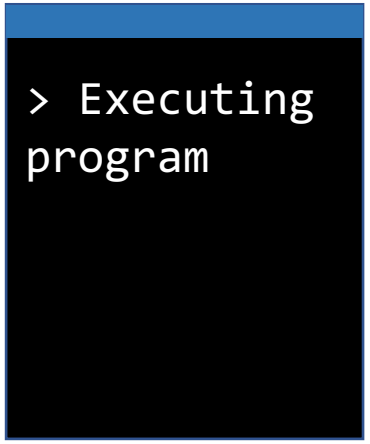
Exploiting bugs in applications

```
void handleReq(char *uname, char * upasswd, char *fname, char *out, int out_size) {
    char passwd [SIZE] , fcontents [SIZE];
    read_password ( uname , passwd , SIZE );
    if (!( authenticate (uname, upasswd, passwd ))) {
        return;
    }
    // inadvertently copying the password to the log file
    send(log_file, passwd, SIZE );

    read_file(fname, fcontents, SIZE );

    //( out_size > SIZE ) can leak passwd to out
    memcpy(out, fcontents, out_size );


    sprintf(out + SIZE, fmt, "Request complete");
    return;
}
```

Exploiting bugs in applications

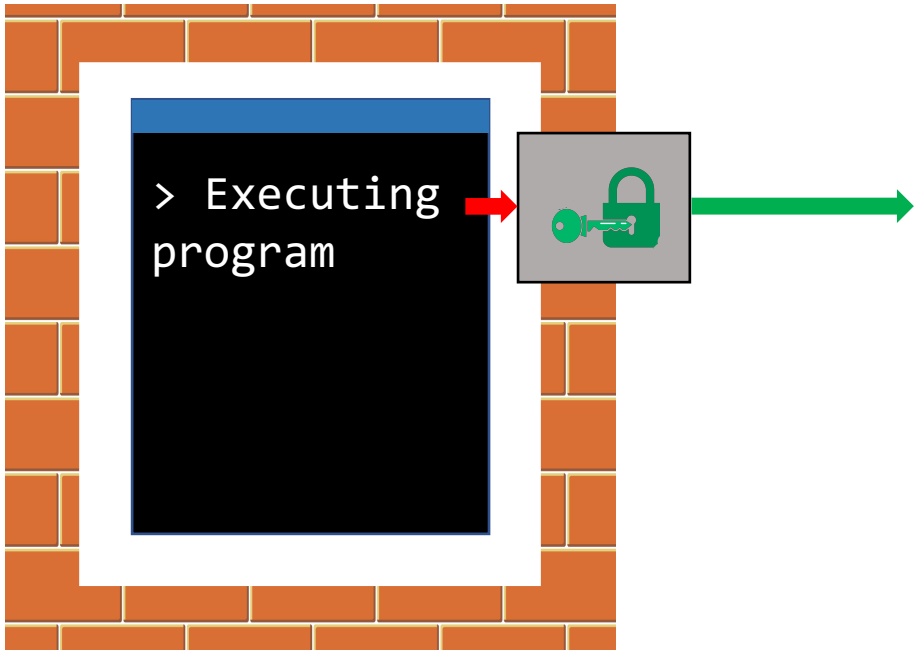
```
void handleReq(char *uname, char * upasswd, char *fname, char *out, int out_size) {
    char passwd [SIZE] , fcontents [SIZE];
    read_password ( uname , passwd , SIZE );
    if (!( authenticate (uname, upasswd, passwd ))) {
        return;
    }
    // inadvertently copying the password to the log file
    send(log_file, passwd, SIZE );

    read_file(fname, fcontents, SIZE );

    //( out_size > SIZE ) can leak passwd to out
    memcpy(out, fcontents, out_size );

    //a bug in the fmt string can print stack contents
    sprintf(out + SIZE, fmt, "Request complete");
    return;
}
```

Securing data from bugs

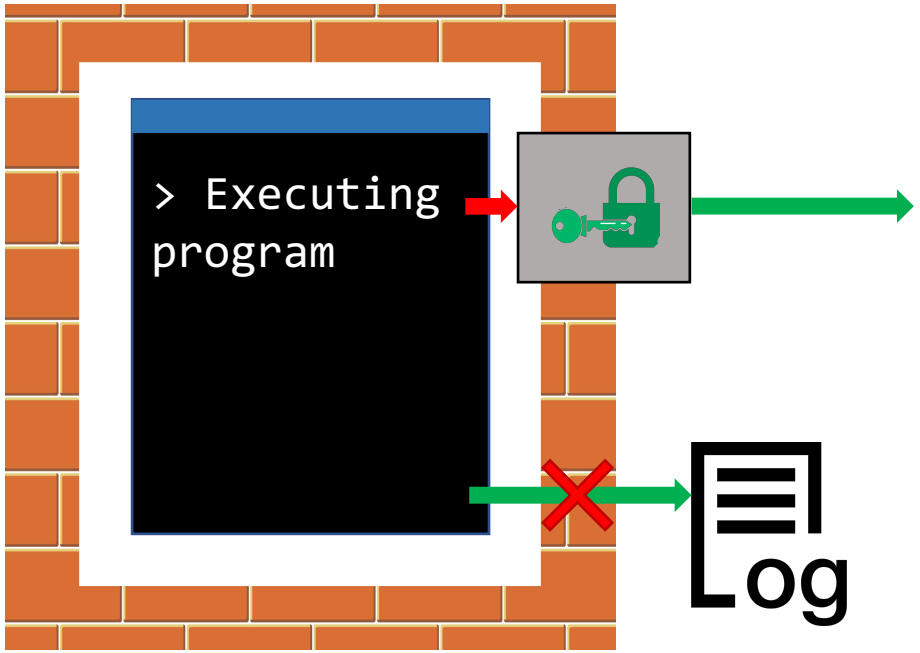


Securing data from bugs

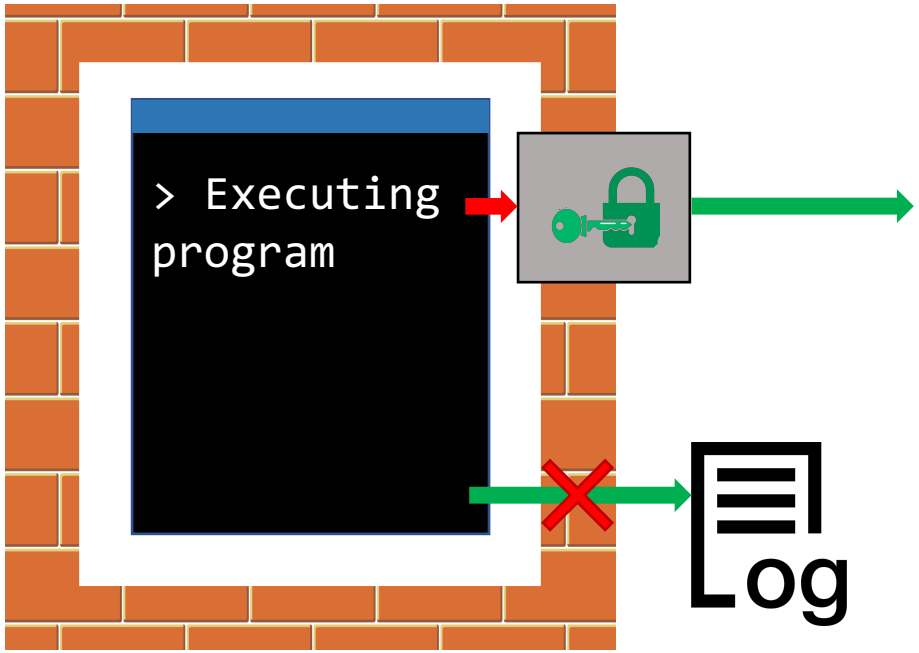
Securing data from bugs



Securing data from bugs

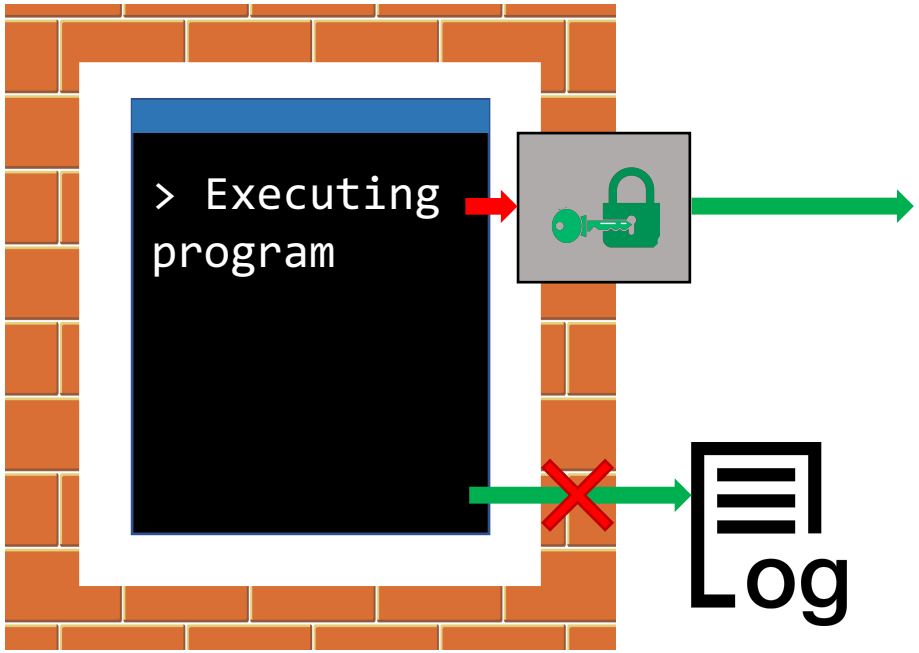


Securing data from bugs



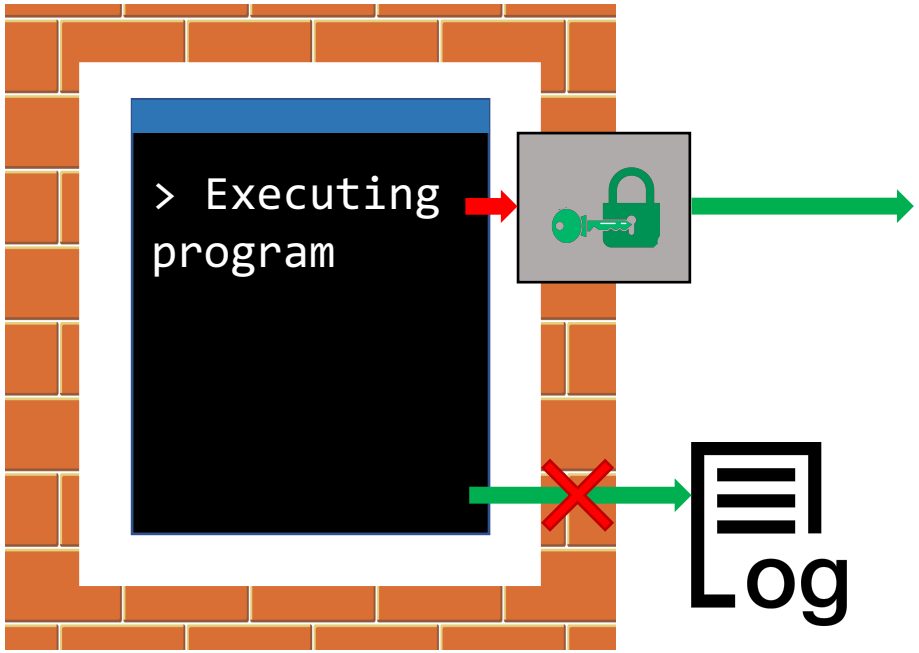

```
int averageSalary(int salaries[], int count) {
    int totalSalary = 0;
    for (int i = 0; i < count; i++)
        totalSalary += salaries[i];
    int average = totalSalary / count;
    return average;
}
```

Securing data from bugs




```
int averageSalary(int salaries[], int count) {
    int totalSalary = 0;
    for (int i = 0; i < count; i++)
        totalSalary += salaries[i];
    int average = totalSalary / count;
    return average;
}
```

Securing data from bugs




```
int averageSalary(int salaries[], int count) {
    int totalSalary = 0;
    for (int i = 0; i < count; i++)
        totalSalary += salaries[i];
    int average = totalSalary / count;
    return average;
}
```

Information Flow Control / Taint Tracking

Information Flow Control

Information Flow Control

- Static analysis techniques
  - Brown *et al*. (2016)– “How to build static checking systems using orders of magnitude less code”
  - Rocha *et al*. (2013)– “Hybrid static-runtime information flow and declassification enforcement”

Information Flow Control

- Static analysis techniques
  - Brown *et al*. (2016)– “How to build static checking systems using orders of magnitude less code”
  - Rocha *et al*. (2013)– “Hybrid static-runtime information flow and declassification enforcement”
- Dynamic Taint tracking
  - Newsome *et al*. (2005)– “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software”
  - Henderson *et al*. (2017)– “Decaf: A platform-neutral whole-system dynamic binary analysis platform”

Information Flow Control

- Static analysis techniques
  - Brown *et al*. (2016)– “How to build static checking systems using orders of magnitude less code”
  - Rocha *et al*. (2013)– “Hybrid static-runtime information flow and declassification enforcement”
- Dynamic Taint tracking
  - Newsome *et al*. (2005)– “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software”
  - Henderson *et al*. (2017)– “Decaf: A platform-neutral whole-system dynamic binary analysis platform”
- Complete memory and type safety
  - Necula *et al*. (2005)– “Ccured: type-safe retrofitting of legacy software”
  - Nagarkatte *et al*. (2009)– “SoftBound: highly compatible and complete spatial memory safety for C”

Introducing ConfLLVM

Taint tracking in ConfLLVM

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Tracking taints of

1. Registers
2. Bytes in memory

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Tracking taints of

1. Registers
2. Bytes in memory

If we know the taint of each byte we load from memory, we do not have to track the taints of the registers at runtime!

Taint tracking in ConfLLVM

```
int bar (int * a, int * b, int * c, int i) {
     int b_val = b[i];
     int c_val = c[i];
     int a_val = b_val + c_val;
     a[i] = a_val;
     return a_val;
}
```

```
r1 = b + i          ;
r2 = load [r1]      ; b_val = b[i]
r3 = c + i          ;
r4 = load [r3]      ; c_val = c[i]
r5 = add, r4, r2    ; a_val = b_val + c_val
r6 = a + i          ;
store r5, [r6]      ; a[i] = a_val
ret r5              ; return a_val
```

Tracking taints of

1. ~~Registers~~
2. Bytes in memory

If we know the taint of each byte we load from memory, we do not have to track the taints of the registers at runtime!

Inferring taints of registers statically

```
/* bar.h */
int bar (int *, int*, int*, int);


/* bar.c */
int bar (int *a, int *b, int *c, int i) {
    int b_val = a[i];
    int c_val = b[i];
    int a_val = b_val + c_val;
    a[i] = a_val;
    return a_val;
}
```

Inferring taints of registers statically

```
/* bar.h */
private int bar (private int *, private
int*, int*, int);


/* bar.c */
int bar (int *a, int *b, int *c, int i) {
    int b_val = a[i];
    int c_val = b[i];
    int a_val = b_val + c_val;
    a[i] = a_val;
    return a_val;
}
```

Inferring taints of registers statically

```
/* bar.h */
private int bar (private int *, private
int*, int*, int);


/* bar.c */
private int bar (private int *a, private
int *b, int *c, int i) {
    private int b_val = a[i];
    int c_val = b[i];
    private int a_val = b_val + c_val;
    a[i] = a_val;
    return a_val;
}
```

Inferring taints of registers statically

```
/* bar.h */
private int bar (private int *, private
int*, int*, int);


/* bar.c */
private int bar (private int *a, private
int *b, int *c, int i) {
    private int b_val = a[i];
    int c_val = b[i];
    private int a_val = b_val + c_val;
    a[i] = a_val;
    return a_val;
}
```

```
/* foo.c */

int foo() {
    private int a[16];
    ...
    bar(a, ..., 1024);
}
```

Inferring taints of registers statically

```
/* bar.h */
private int bar (private int *, private
int*, int*, int);


/* bar.c */
private int bar (private int *a, private
int *b, int *c, int i) {
    private int b_val = a[i];
    int c_val = b[i];
    private int a_val = b_val + c_val;
    a[i] = a_val;
    return a_val;
}
```

```
/* foo.c */

int foo() {
    private int a[16];
    ...
    bar(a, ..., 1024);
}
/* bar.S */
r1 = b + i

r2 = load [r1]
r3 = c + i

r4 = load [r3]
r5 = add, r4, r2
r6 = a + i

store r5, [r6]
```

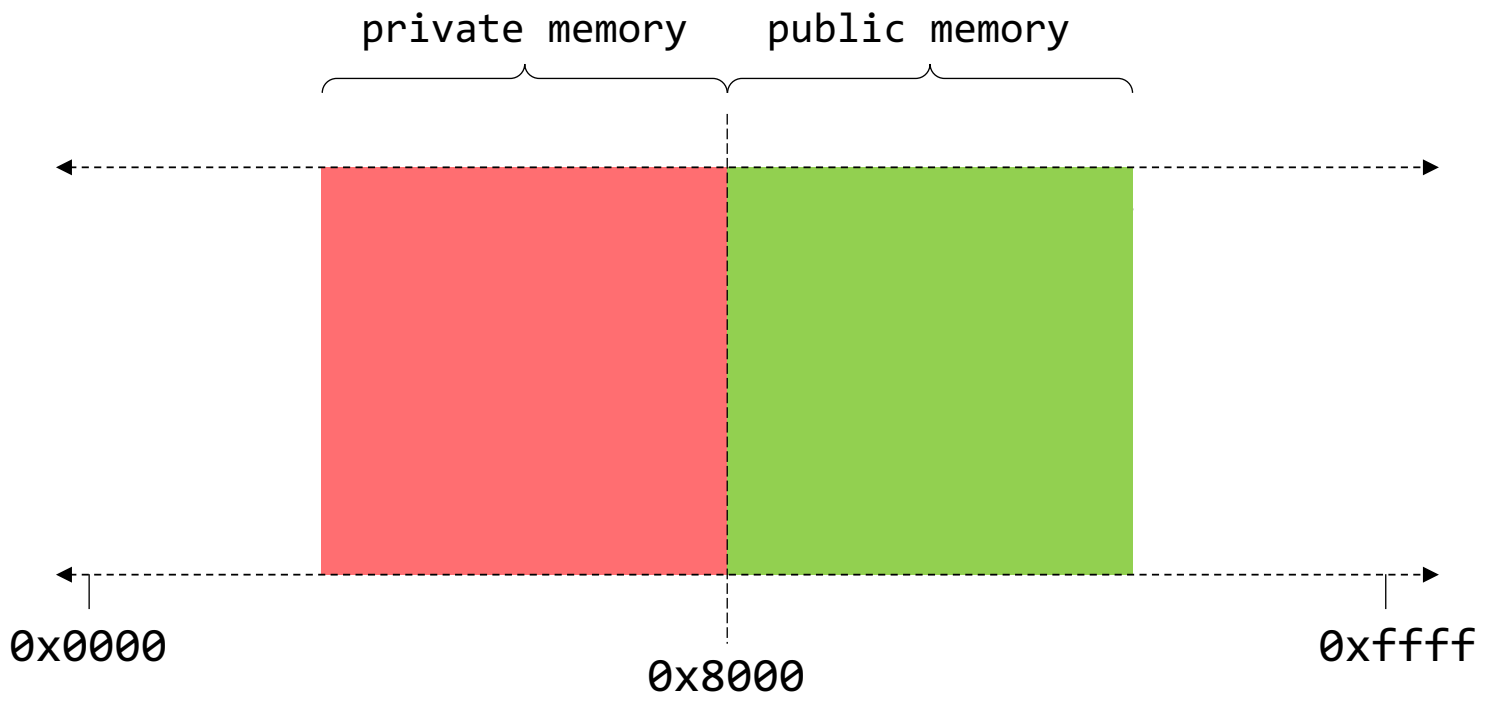
Inferring taints of registers statically

```
/* bar.h */
private int bar (private int *, private
int*, int*, int);


/* bar.c */
private int bar (private int *a, private
int *b, int *c, int i) {
    private int b_val = a[i];
    int c_val = b[i];
    private int a_val = b_val + c_val;
    a[i] = a_val;
    return a_val;
}
```

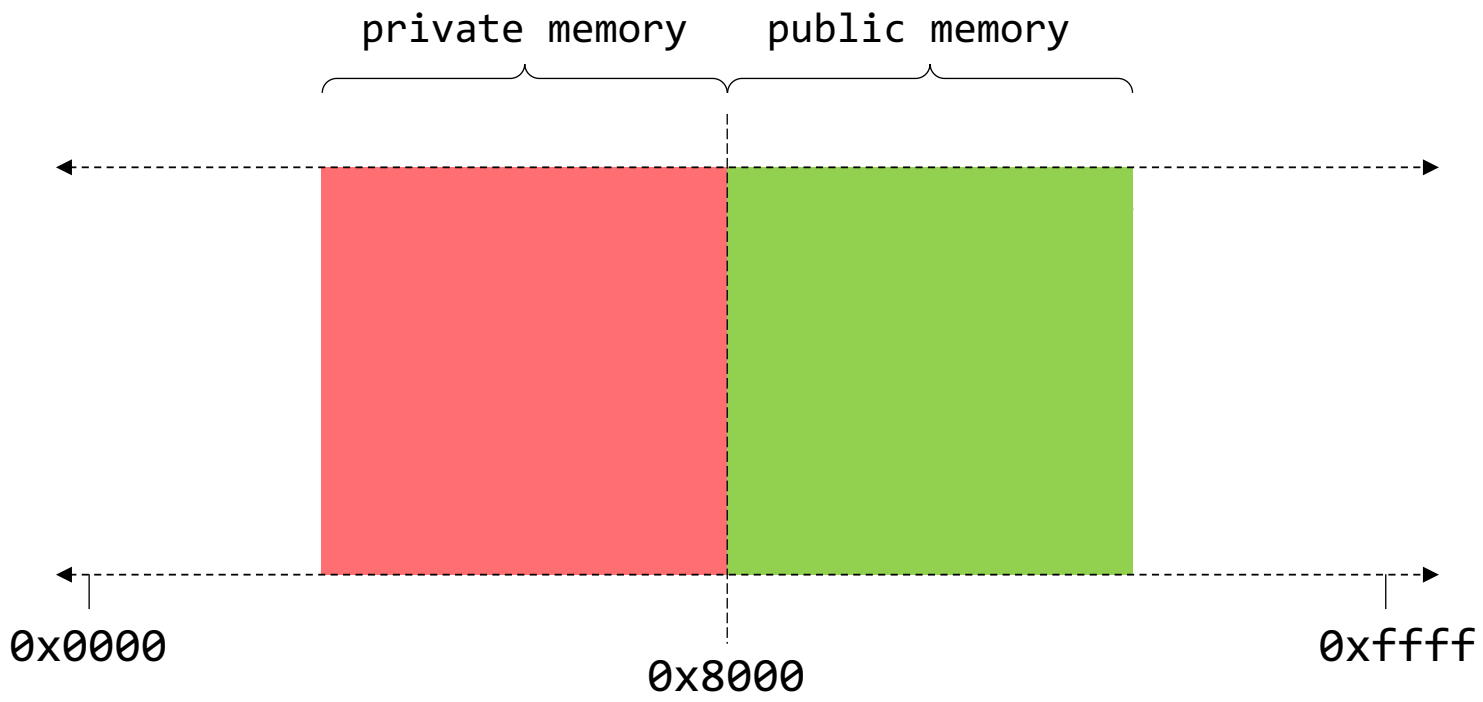
```
/* foo.c */

int foo() {
    private int a[16];
    ...
    bar(a, ..., 1024);
}
/* bar.S */
r1 = b + i
check_taint_private([r1])
r2 = load [r1]
r3 = c + i
check_taint_public([r3])
r4 = load [r3]
r5 = add, r4, r2
r6 = a + i
set_taint_private([r6])
store r5, [r6]
```

Improving memory checks

Improving memory checks



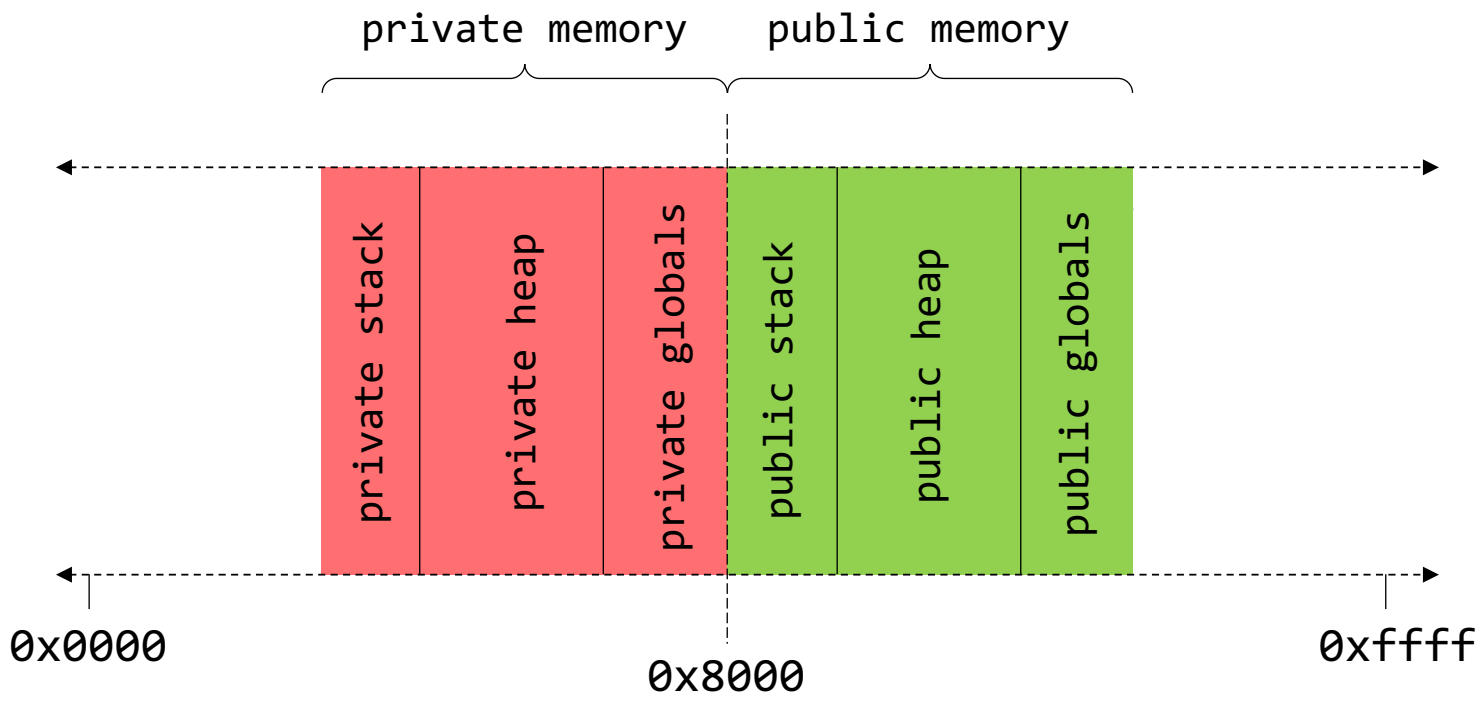
Improving memory checks



check_taint_private([x]) = assert(x < 0x8000);

check_taint_public([x]) = assert(x >= 0x8000);

Improving memory checks



check_taint_private([x]) = assert(x < 0x8000);

check_taint_public([x]) = assert(x >= 0x8000);

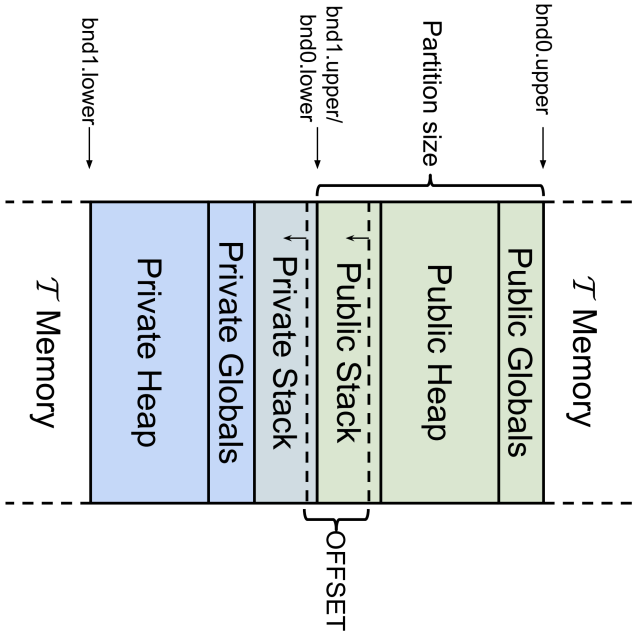
Improving memory checks

Improving memory checks

- Intel MPX
  - `bnd0 – bnd3` – 128 bit registers to hold upper and lower bounds
  - `bndcu [x], bnd0` – check if the address [x] is lower than the upper bound stored in bnd0
  - `bndcl [x], bnd0` – check if the address [x] is greater than or equal to the lower bound stored in bnd0

Improving memory checks

- Intel MPX
  - `bnd0 – bnd3` – 128 bit registers to hold upper and lower bounds
  - `bndcu [x], bnd0` – check if the address [x] is lower than the upper bound stored in bnd0
  - `bndcl [x], bnd0` – check if the address [x] is greater than or equal to the lower bound stored in bnd0




```
check_taint_private([x]) = bndcu [x], bnd1
                           bndcl [x], bnd1

check_taint_public ([x]) = bndcu [x], bnd0
                           bndcl [x], bnd0

Public local load  -   mov   rax, [rsp + 8]

Private local load -   mov   rax, [rsp + 8 – OFFSET]
```

Improving memory checks

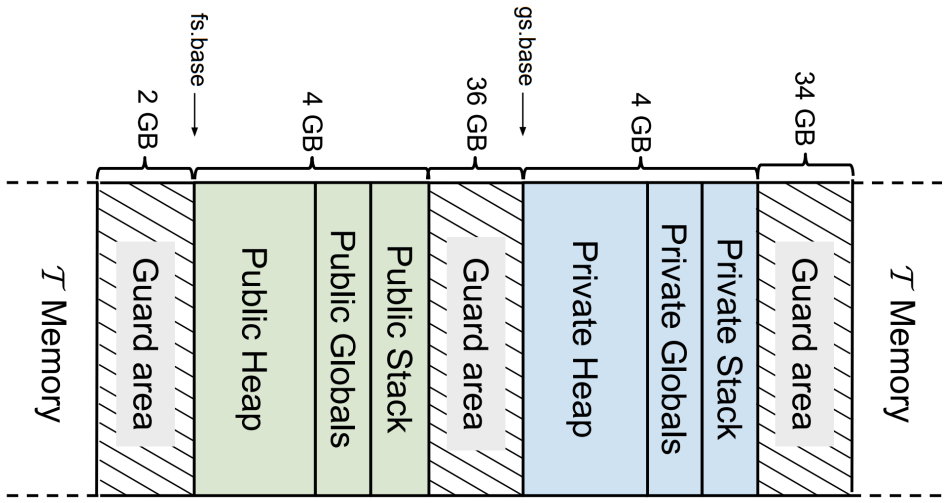
Improving memory checks

- Intel X64 segment registers
  - `gs and fs segment registers` – user can control gs.base and fs.base
  - `mov eax, gs: [ebx]` – loads [gs.base + ebx] into eax;

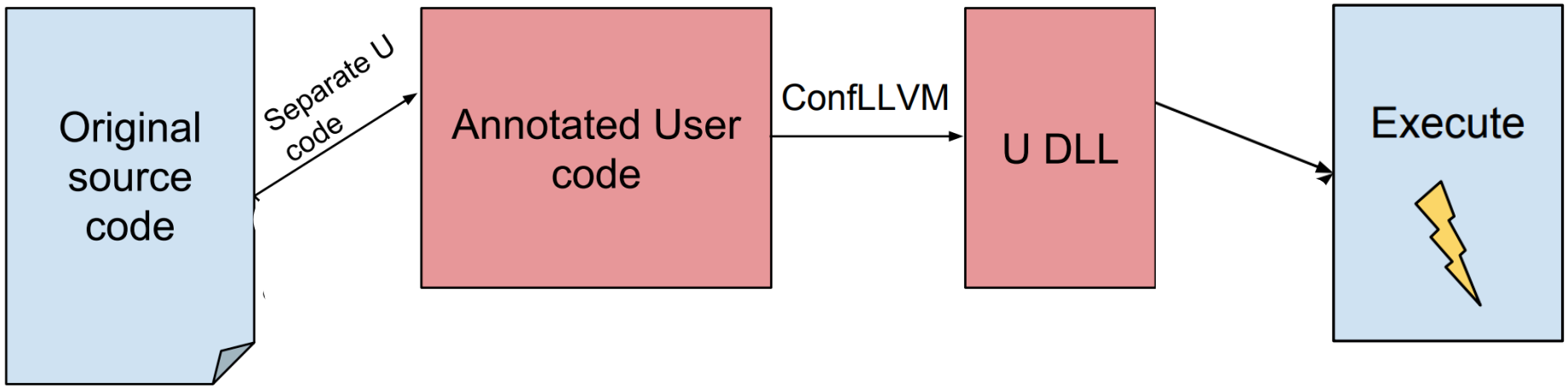
Improving memory checks

- Intel X64 segment registers
  - `gs and fs segment registers` – user can control gs.base and fs.base
  - `mov eax, gs: [ebx]` – loads [gs.base + ebx] into eax; checks if ebx < gs.limit

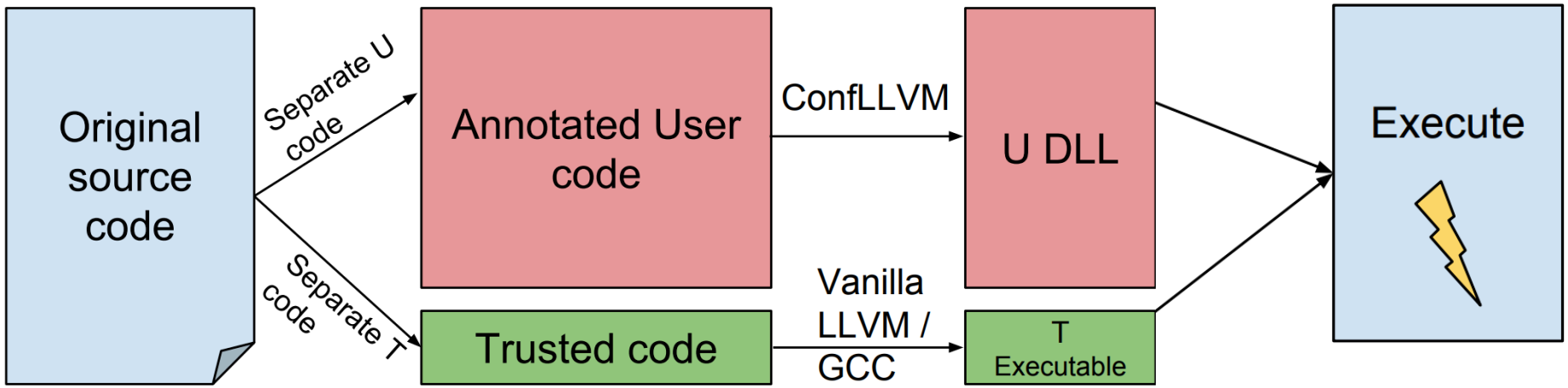
  - `mov rax, [rbx + rcx * 8]` → `mov rax, gs:[ebx + ecx * 8] // Private load`
  - `mov rax, [rbx + rcx * 8]` → `mov rax, fs:[ebx + ecx * 8] // Public load`
  - `mov [rbx + rcx * 8], rax` → `mov gs:[ebx + ecx * 8], rax // Private store`
  - `mov [rbx + rcx * 8], rax` → `mov fs:[ebx + ecx * 8], rax // Public store`



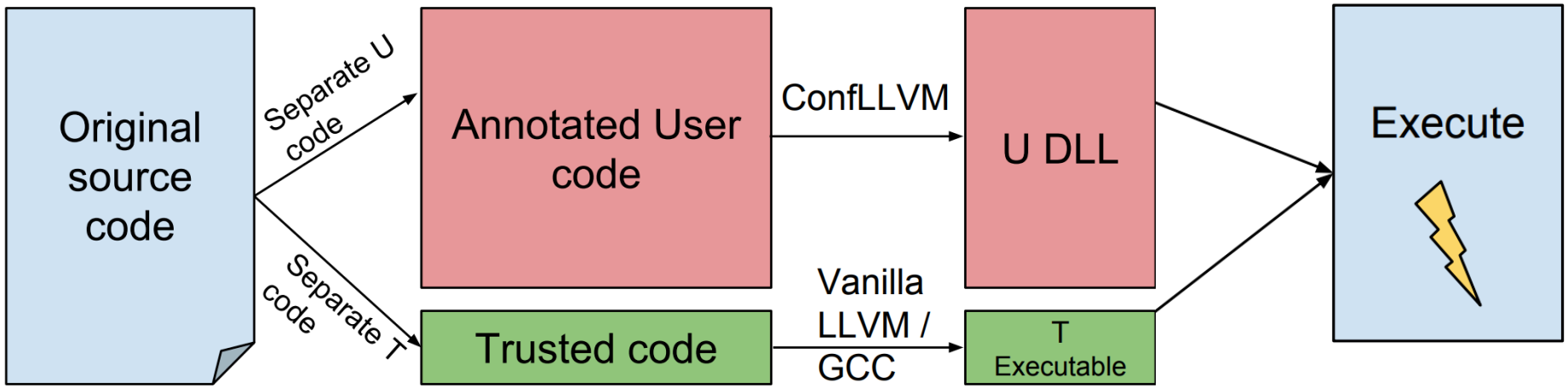
Trusted code



Trusted code



Trusted code



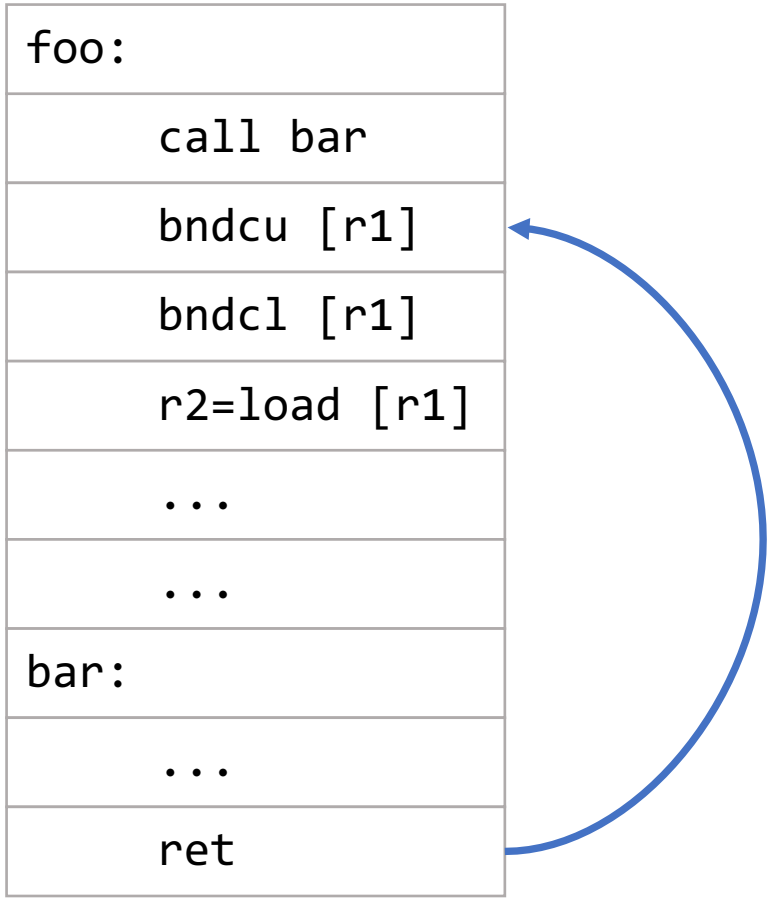

```
void encrypt (private char * plain_text, int plain_text_size, char * cipher_text, int
cipher_text_size);

void decrypt (char * cipher_text, int cipher_text_size, private char * plain_text, int
plain_text_size);
```

Control Flow Integrity

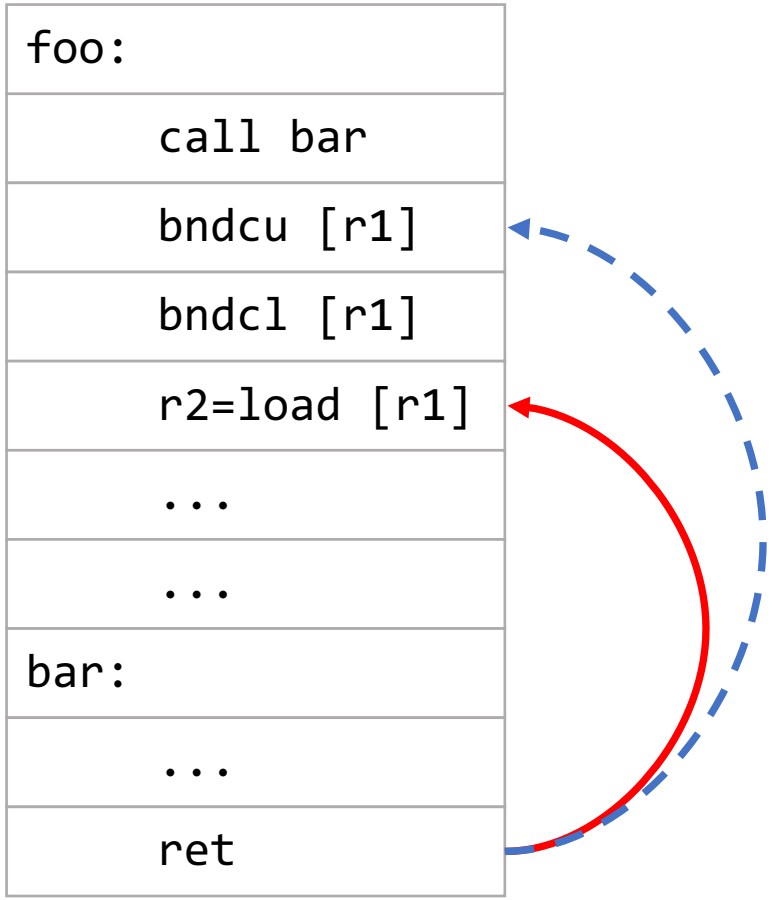
Control Flow Integrity

```
foo:
      call bar
      bndcu [r1]
      bndcl [r1]
      r2=load [r1]
      ...
      ...
bar:
      ...
      ret
```

Control Flow Integrity



Control Flow Integrity



Buffer overflow in bar corrupts the return value!

Control Flow Integrity

- Potential threats –
  - Indirect jumps
  - Indirect calls
  - Call return sites

Control Flow Integrity

- Potential threats –
  - Indirect jumps
  - Indirect calls
  - Call return sites

- Need to check –
  - If the target is a valid jump site
  - The target expects the same taints on the registers as current registers

Control Flow Integrity

- Magic Strings! – a sequence of bytes 8 byte long that is not found ANYWHERE in the program executable section
- Insert magic strings before function start and after call instructions

Control Flow Integrity

- Magic Strings! – a sequence of bytes 8 byte long that is not found ANYWHERE in the program executable section
- Insert magic strings before function start and after call instructions

```
foo:
        ...
        callq bar
        #MAGIC_STRING+0001 //private return
          ...

bar:
        ...
        popq %r11
        //Check before return
        cmpq ($r11), #MAGIC_STRING+0001
        jne REPORT_ERROR
        jmpq *(%r11 + 8)
```

Control Flow Integrity

- Magic Strings! – a sequence of bytes 8 byte long that is not found ANYWHERE in the program executable section
- Insert magic strings before function start and after call instructions

```
foo:
        ...
        callq bar
        #MAGIC_STRING+0001 //private return
          ...

bar:
        ...
        popq %r11
        //Check before return
        cmpq ($r11), #MAGIC_STRING+0001
        jne REPORT_ERROR
        jmpq *(%r11 + 8)
```

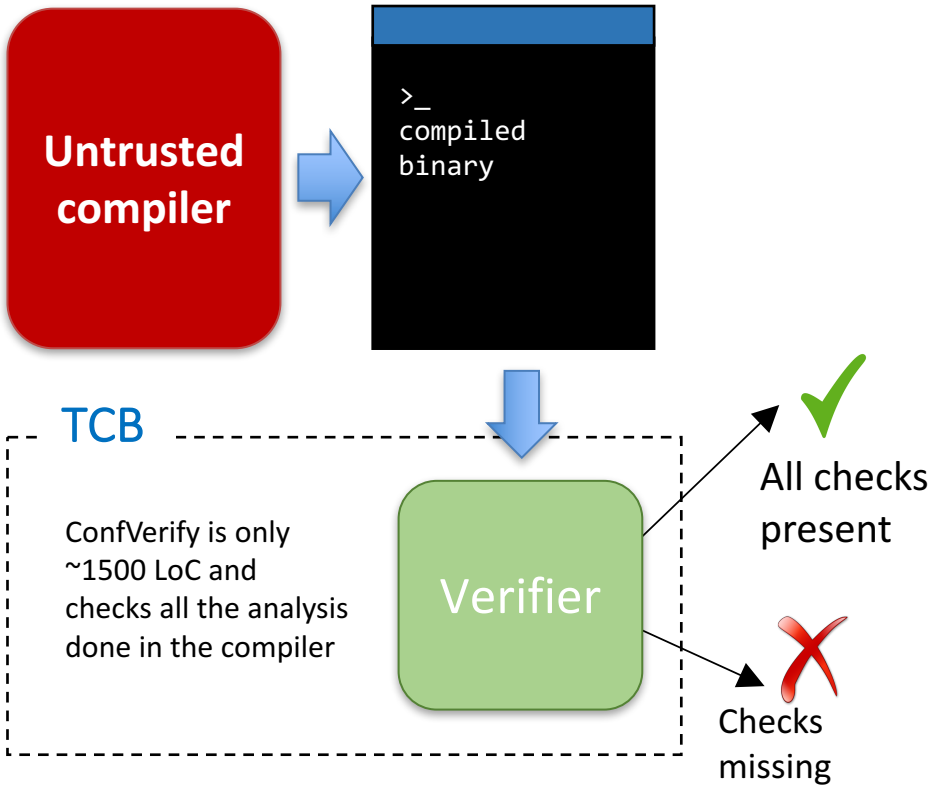
```
foo:
        movq      $bar, %r10
        ...
        // 1010 – taints of args encoded
        cmpq -8(%r10), #MAGIC_STRING+1010
        jne REPORT_ERROR
        callq *%r10
        ...

        // taint of expected args encoded
        #MAGIC_STRING+1010
bar:
        ...
```

Can the compiler be trusted?

- LLVM compiler is over 2.5 million lines of code
- Bugs in the compiler can break all the guarantees

Can the compiler be trusted?

- LLVM compiler is over 2.5 million lines of code
- Bugs in the compiler can break all the guarantees



Evaluations

Evaluations

- Performance overhead – Execution time

Evaluations

- Performance overhead – Execution time
- Memory overhead – None

Evaluations

- Performance overhead – Execution time
- Memory overhead – None
- Programmer overhead
  - Adding annotations to function headers
  - Rewrite some functions to conform to the requirements of the implementation

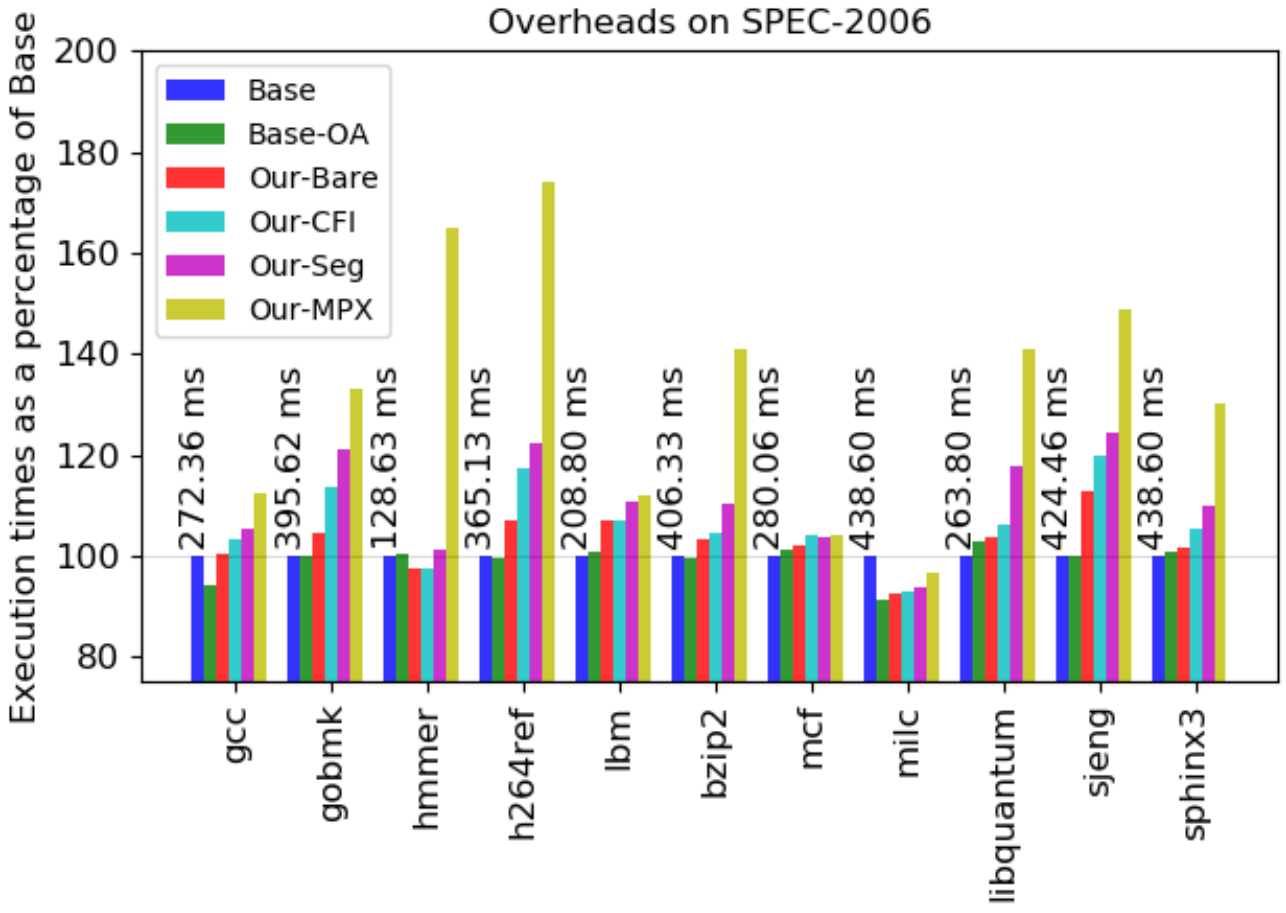
Evaluations

- Performance overhead – Execution time
- Memory overhead – None
- Programmer overhead
  - Adding annotations to function headers
  - Rewrite some functions to conform to the requirements of the implementation

- Applications
  - Spec 2006 C benchmarks
  - OpenLDAP
  - Nginx web server
  - ML prediction inside Intel SGX

Evaluations – spec 2006

- Everything inside the applications is public
- Understand the runtime overheads

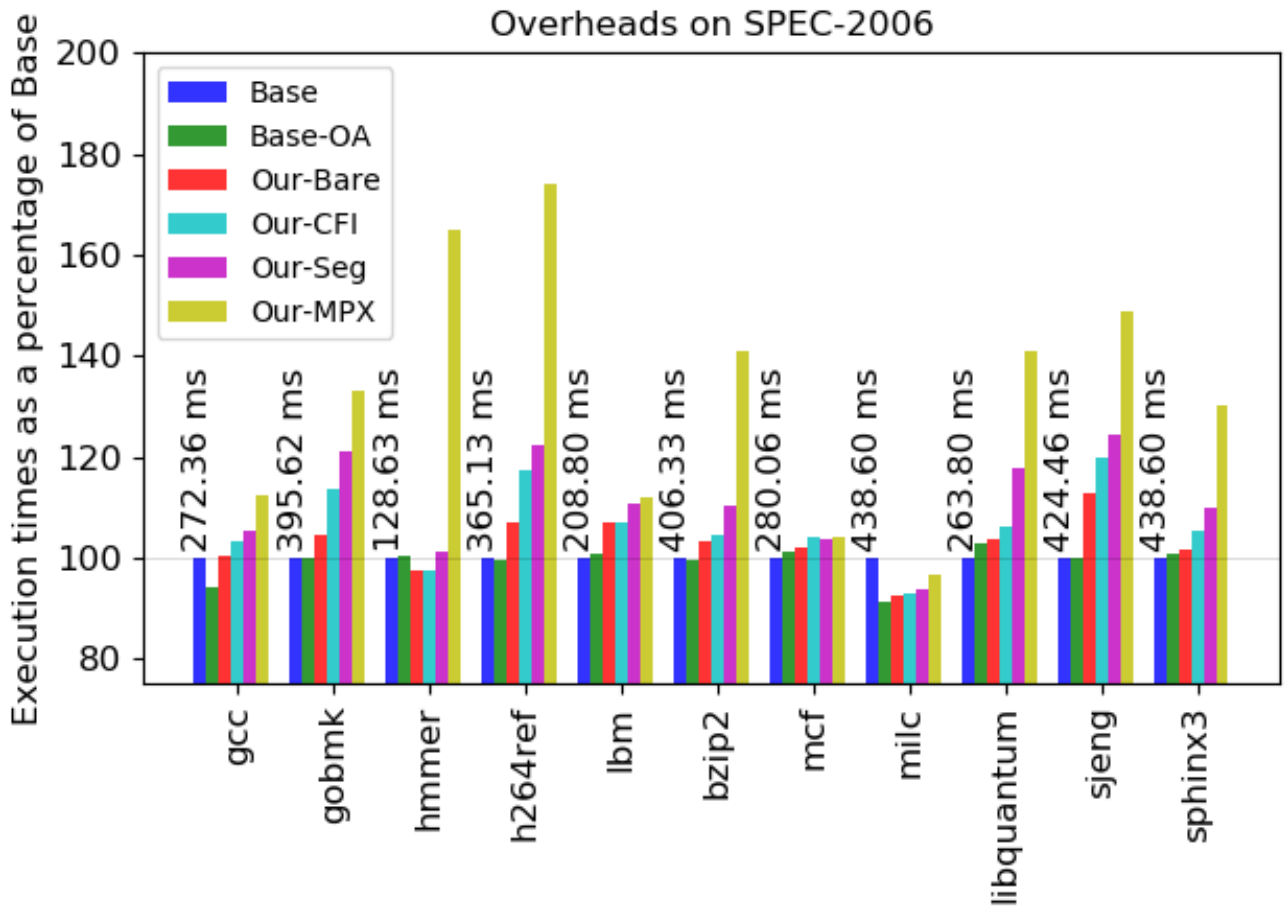


All benchmarks run on a Microsoft Surface Pro-4 Windows 10 machine with an Intel Core i7-6650U 2.20 GHz 64-bit processor with 2 cores (4 logical cores) and 8 GB RAM.

Applications compiled “as is”. No code changes because public annotation is default

Evaluations – spec 2006

- Everything inside the applications is public
- Understand the runtime overheads

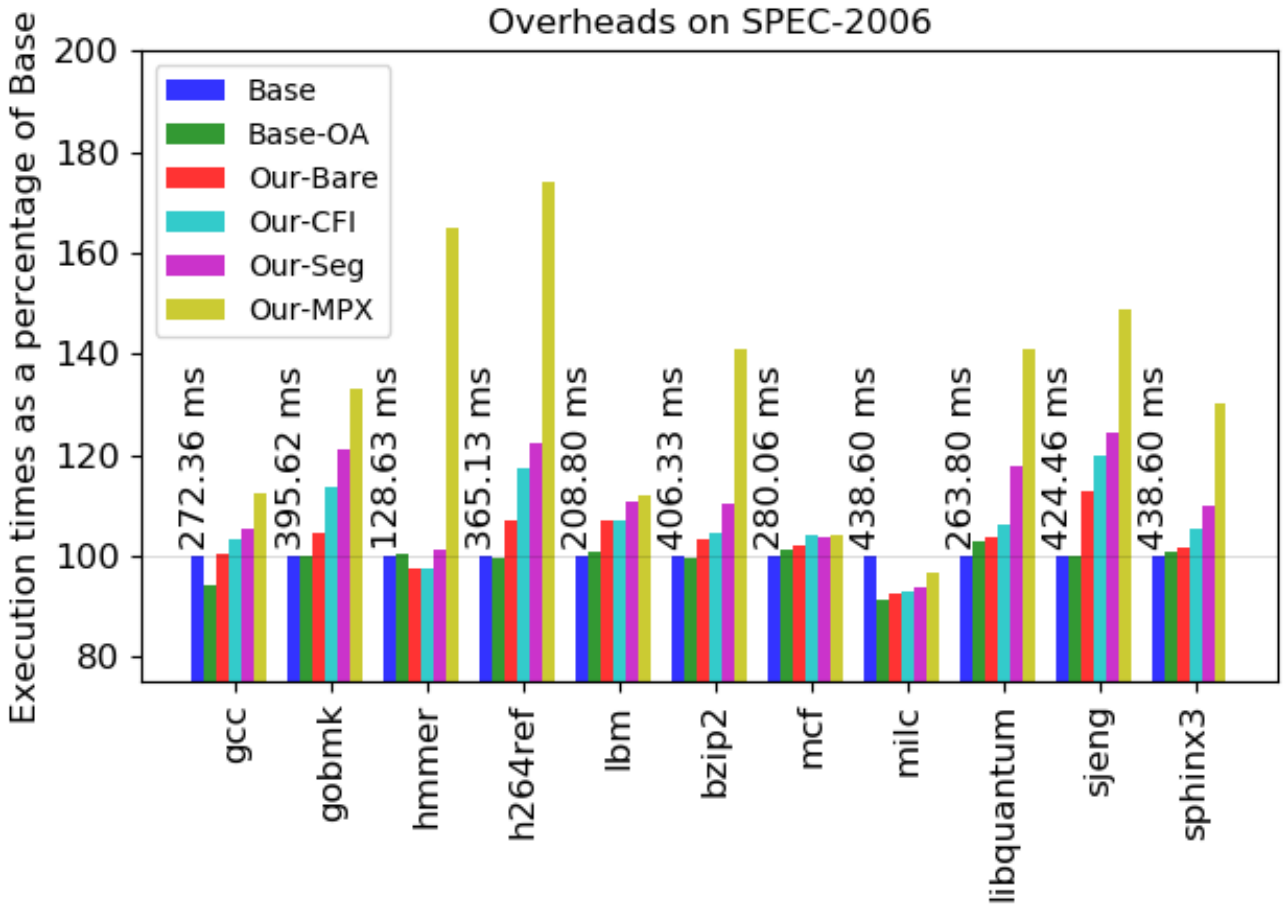


All benchmarks run on a Microsoft Surface Pro-4 Windows 10 machine with an Intel Core i7-6650U 2.20 GHz 64-bit processor with 2 cores (4 logical cores) and 8 GB RAM.

Applications compiled “as is”. No code changes because public annotation is default

Evaluations – spec 2006

- Everything inside the applications is public
- Understand the runtime overheads

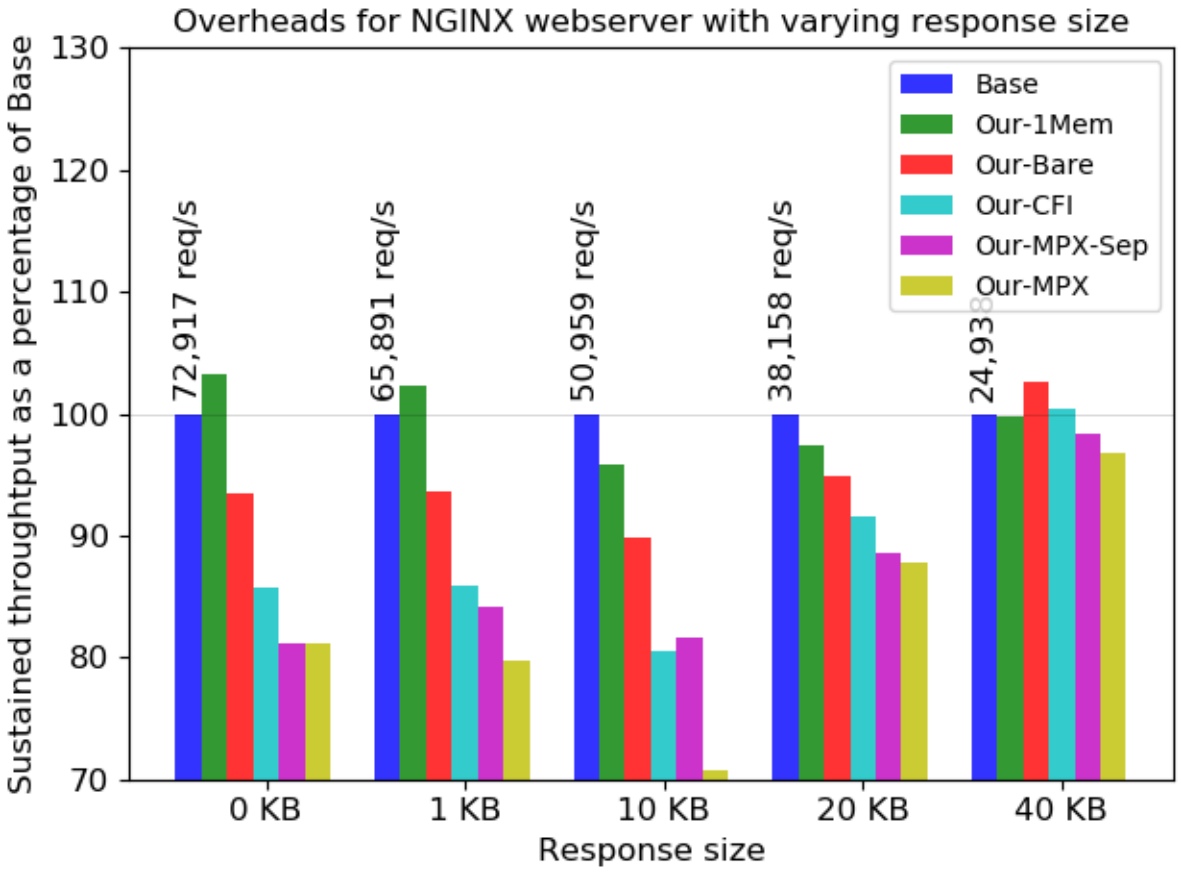


All benchmarks run on a Microsoft Surface Pro-4 Windows 10 machine with an Intel Core i7-6650U 2.20 GHz 64-bit processor with 2 cores (4 logical cores) and 8 GB RAM.

Applications compiled “as is”. No code changes because public annotation is default

Evaluations - NGINX

- Everything except logs marked private

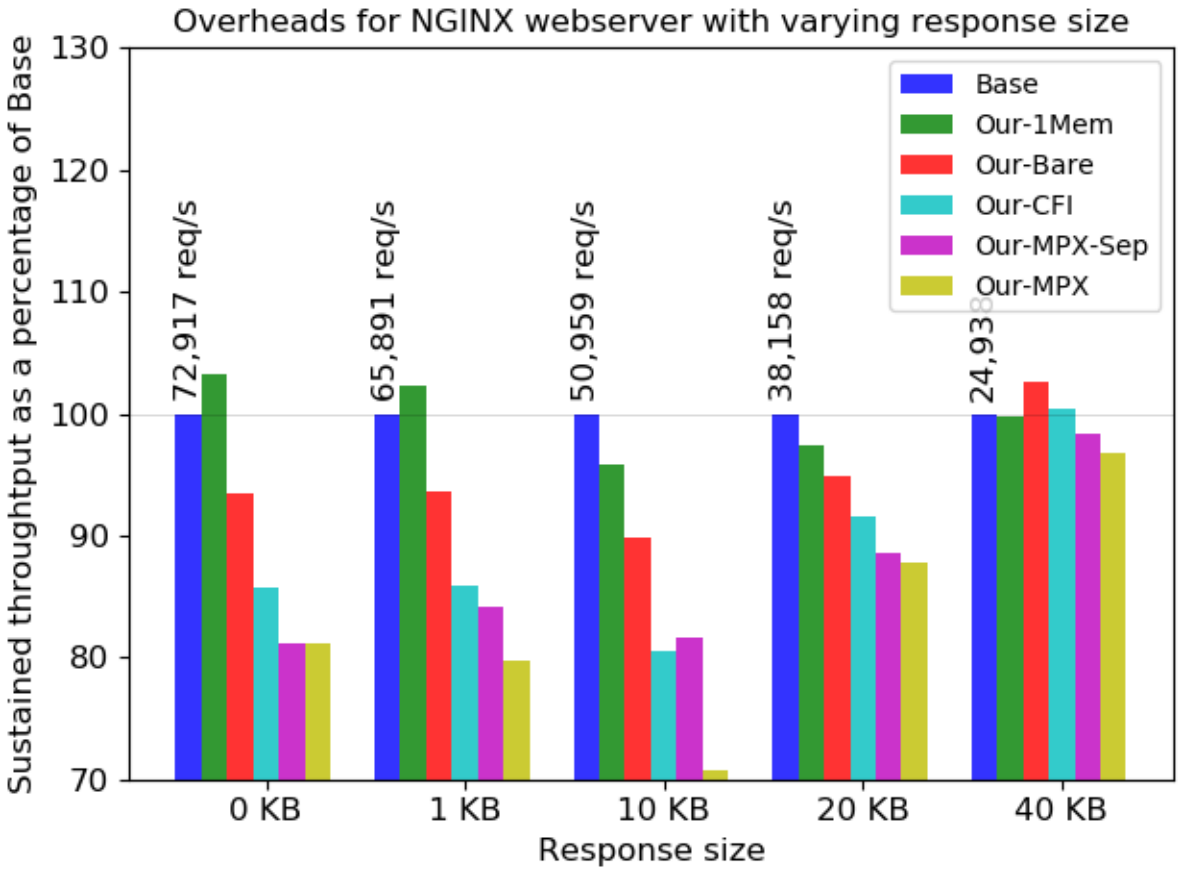


NGINX version 1.13.12 on an Intel Core i7-6700 3.40GHz 64-bit processor with 4 cores (8 logical cores), 32 GB RAM, and a 10 Gbps ethernet card, running Ubuntu v16.04.1 with Linux kernel version 4.13.0-38-generic

Out of the total 124,001 lines of code, 160 lines had to be modified and 138 lines added for the trusted code (encryption + interface with operating system)

Evaluations - Nginx

- User passwords marked as private

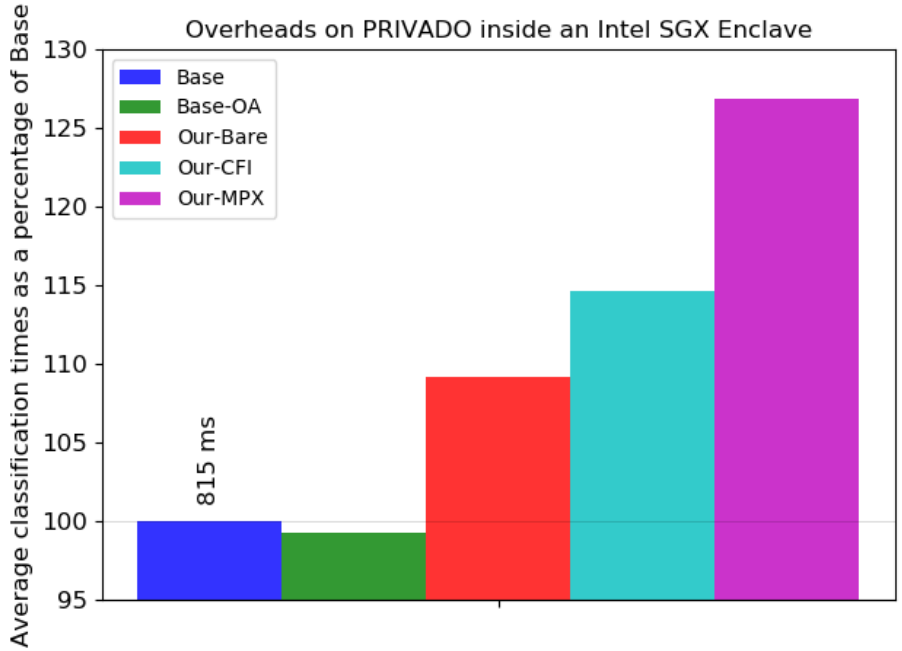


NGINX version 1.13.12 on an Intel Core i7-6700 3.40GHz 64-bit processor with 4 cores (8 logical cores), 32 GB RAM, and a 10 Gbps ethernet card, running Ubuntu v16.04.1 with Linux kernel version 4.13.0-38-generic

Out of the total 124,001 lines of code, 160 lines had to be modified and 138 lines added for the trusted code (encryption + interface with operating system)

Evaluations - ConfLLVM and Intel SGX

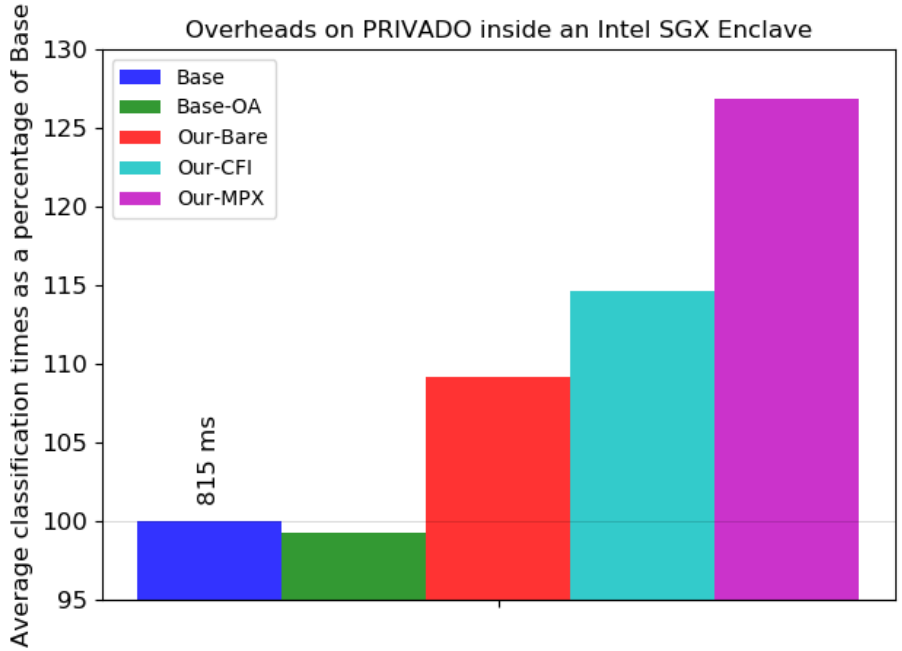
- Image classification algorithm run inside Intel SGX
- All data marked as private including the model and user data
- The classification result declassified and sent out over the network



PRIVADO setup on an Intel Core i7-6700 3.40GHz 64-bit processor with an Ubuntu 16.04 OS (Linux 4.15.0-34-generic kernel)

Evaluations - ConfLLVM and Intel SGX

- Image classification algorithm run inside Intel SGX
- All data marked as private including the model and user data
- The classification result declassified and sent out over the network



PRIVADO setup on an Intel Core i7-6700 3.40GHz 64-bit processor with an Ubuntu 16.04 OS (Linux 4.15.0-34-generic kernel)

Correctness guarantee

- Meta theory proof
- Relies on the invariants checked in ConfVerify
- Proves termination sensitive non interference on private data (assuming that the trusted components are non interfering)

- Proof available as a technical report - https://arxiv.org/abs/1711.11396

Summary

- ConfLLVM a compiler technique for Information Flow Control that uses combination of static and dynamic analysis to ensure that the private data doesn’t leak out through clear text channels
- Novel memory partitioning technique for efficiently checking taints of data in memory
- A lightweight Control Flow Integrity technique that provides just enough guarantees for the IFC property
- Trusted Code Base potentially reduced to ~1500 LoC

https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-6-115 -- genomic data image