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

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

Confidentiality attacks due to bugs in programs

```
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;
    }
    read_file ( fname , fcontents , SIZE );
    // ( out_size > SIZE ) can leak passwd to out
    memcpy ( out , fcontents , out_size );
    send(socket, out, out_size);
    …
}
```

- Bugs in low-level C programs dealing with confidential data can be exploited by active attackers to steal information
- Example: Heartbleed attack on OpenSSL discovered in 2014
- ***Information Flow Control: Ensure that private data and data derived from it is never sent out on a public channel***

Existing solutions

a. Static analysis and software verification
   - C lacks memory and type safety
   - Aliasing and other runtime information makes the analysis incomplete

b. Dynamic instrumentation and taint tracking
   - Extremely high overheads

c. Program in type and memory safe subsets of C
   - Very restrictive programming model
   - No backward compatibility with existing libraries

We present the first end-to-end practical compiler-based scheme to enforce confidentiality in C programs even in the presence of active, low-level attacks

Our methodology – A mixed static and dynamic approach

```
read_password(
char* uname,
private char* passwd,
size_t SIZE
);

void encrypt(
private char*,
private char*,
char*,
unsigned int
);
```

Annotated source code

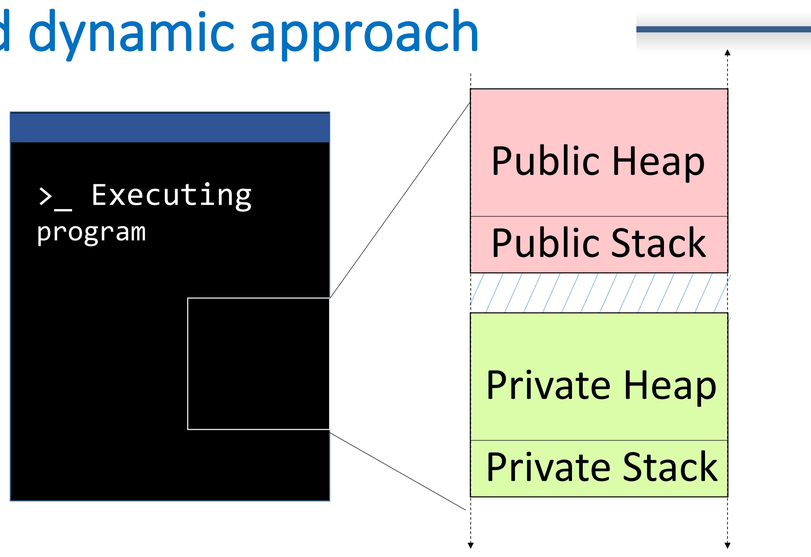
Annotations are not trusted

Instrumenting compiler (LLVM-based)

Static analysis for information flow based on annotations

>_ Executing program

Instrumented executable to assert the annotations



Partitioned memory layout for public and private data

- Static analysis helps eliminate checks on register moves and arithmetic operations
- Partitioning memory coarsely into regions reduces memory taint checking to mere bounds checks
- Novel instrumentation on indirect jumps to ensure data confidentiality despite control-flow hijacks

Runtime checks and optimizations

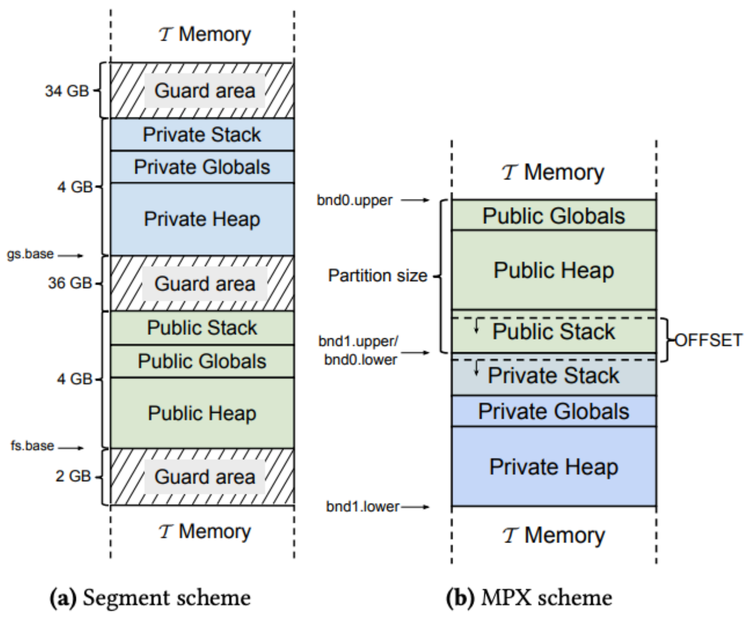
```
bndcl [rsp+8], bnd0
Bndcu [rsp+8], bnd0
load [rsp+8], rax
bndcu [16+rsp], bnd1
bndcl [16+rsp], bnd1
load [16+rsp], rbx
// No check before add
add rbx, rax
bndcu [24+rsp], bnd0
bndcl [24+rsp], bnd0
store rax, [24+rsp]
```

a) MPX scheme

```
// No check before load
load fs:[8+esp], rax

// No check before load
load gs:[16+esp], rbx

// No check before add
add rbx, rax

// No check before store
store rax, fs:[24+esp]
```

b) Segment register scheme

Checks before loads and stores



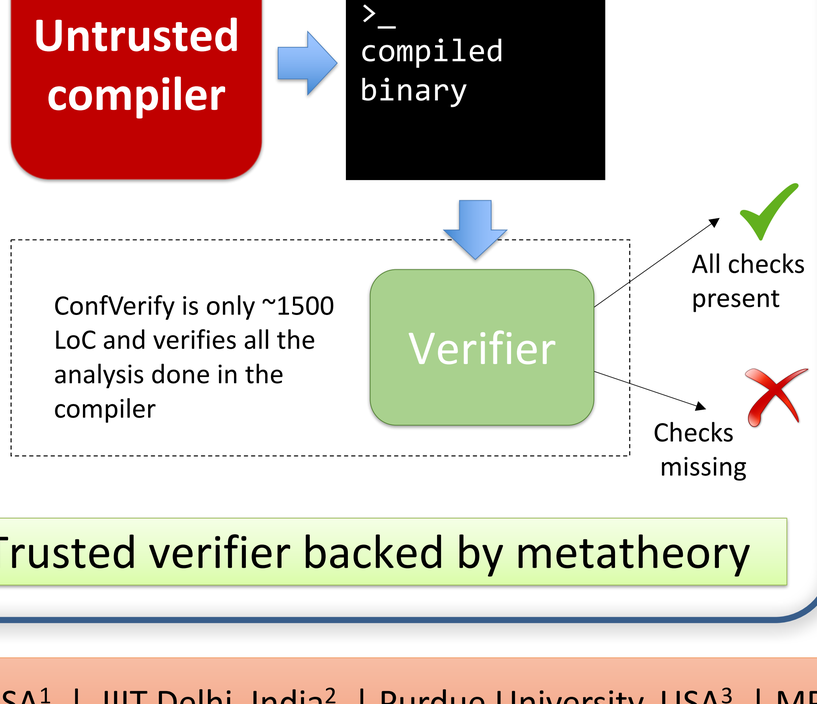
Memory layout to simplify checks

```
foo:
  ...
  // check before indirect call
  check_magic_string(r1, #MAGIC+1010)
  call r1
  #MAGIC+0001 //private return
  ...

  #MAGIC+1010
bar:
  ...
  pop r11
  //Check before return
  check_magic_string(r11, #MAGIC+0001)
  jmp r11+8
```

Checks for protecting function returns and indirect calls

Reducing the TCB

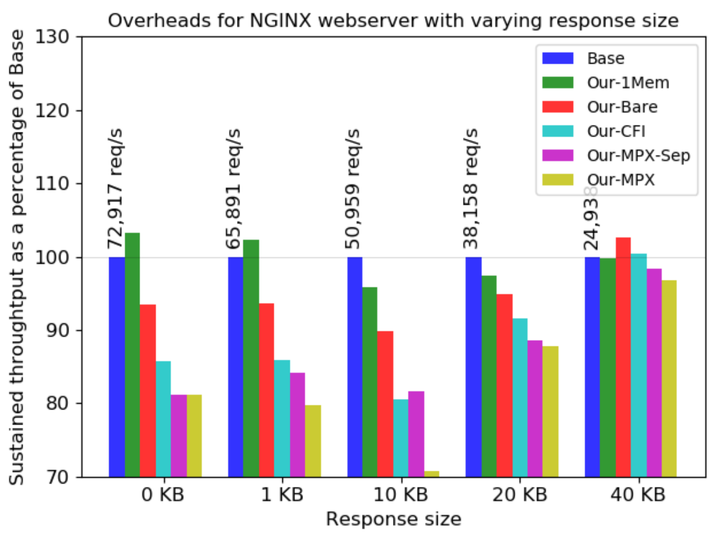
LLVM compiler is 2.5+ million LoC. Bugs in the compiler can manifest as leaks in programs



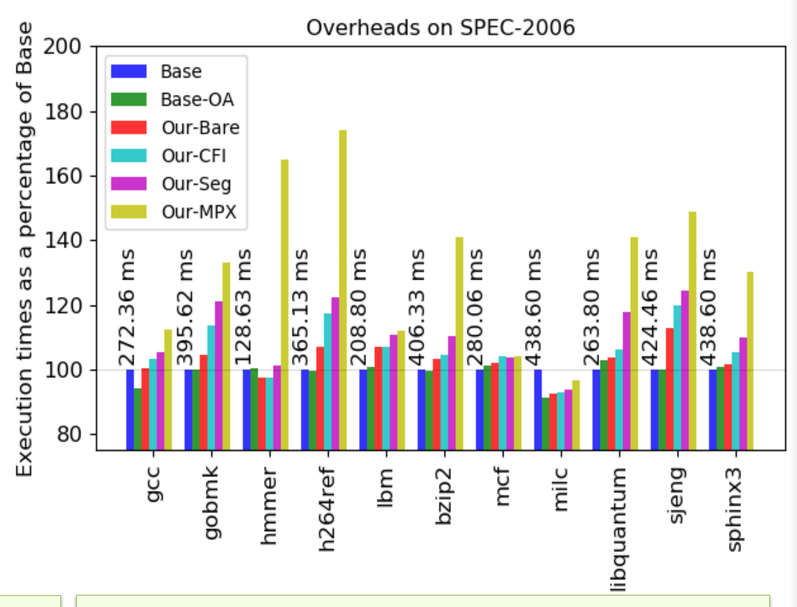
Trusted verifier backed by metatheory

Performance evaluation

Evaluation of runtime overhead and code changes required



Protecting NGINX webserver from leaking sensitive file data into logs (298 LoC changed out of 124,001 LoC)



Spec 2006 compiled with ConfLLVM to measure overhead from dynamic instrumentation

MIT, USA[1] | IIIT Delhi, India[2] | Purdue University, USA[3] | MPI-SWS, Germany[4] | Microsoft Research, India[5] | CISPA, Saarland University, Germany[6] | AMD, India[7]