Glamdring: Automatic Application Partitioning for Intel SGX

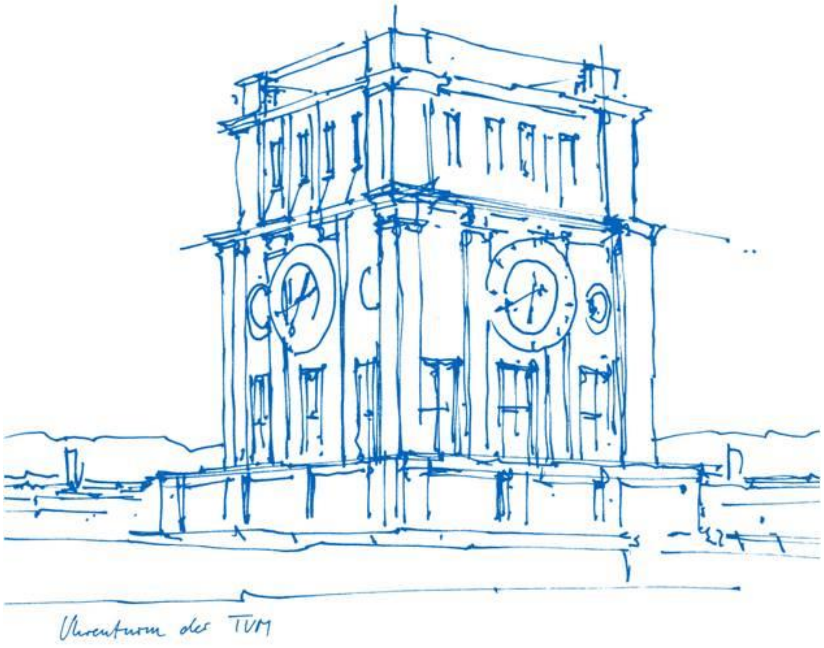
Jokubas Trinkunas

Technical University of Munich

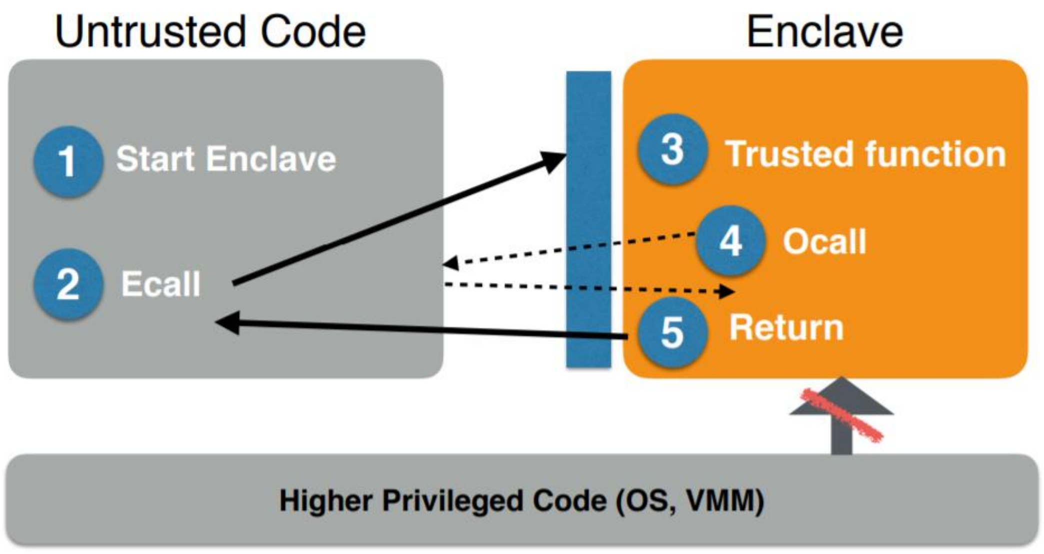
Department of Informatics

Chair for IT Security

05.12.2018



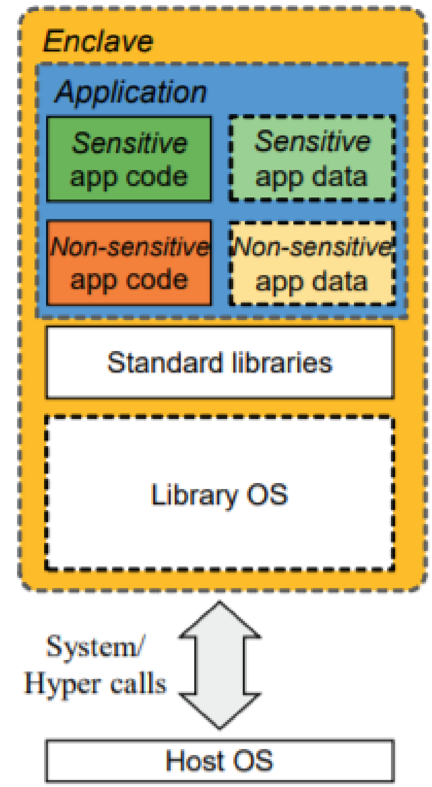
Revision: Intel SGX



- During the execution of the trusted function **multiple ocalls are needed** (to access functionality which is not available in the enclave, for example syscalls)
- Enclave crossing → significant **performance penalty** (enclave state has to be saved and restored)

Design alternatives: Complete enclave interface

- Approach considered by: HAVEN and Graphene
- SCONE: Similar approach without LibOS, but with enhanced C library instead

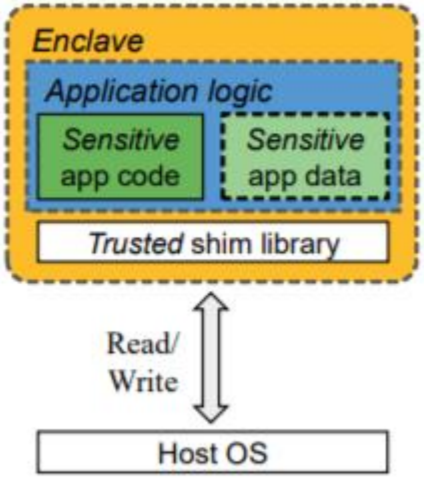


(a) Complete enclave interface

- Pros:
  - Run unmodified applications (low dev. effort)
- Cons:
  - Large TCB (both security-sensitive and insensitive application code and data are inside the enclave + additional libraries)

Design alternatives: Predefined restricted enclave interface

- Approach considered by: VC3 (Verifiable Confidential Cloud Computing)
- Protects distributed map/reduce computations using enclaves (only read/write operations)



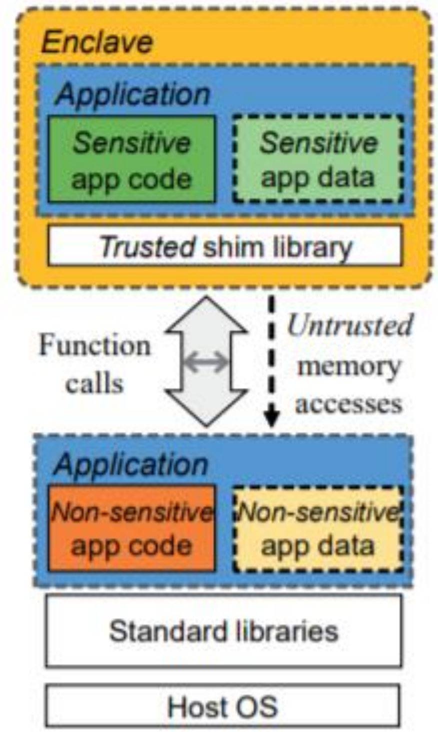
(b) Predefined enclave interface

- Pros:
  - Smaller TCB compared to previous approach
- Cons:
  - Limited applicability (predefined interface → specific applications only, e.g. Hadoop with VC3)

Design alternatives: Application-specific enclave interface

- **Idea**: Only a **subset** of code handles **sensitive data**, other code is not security-sensitive
- Past work has shown that **partitioning can be done manually**
  → Glamdring goal: **automatic** partitioning!

- Pros:
  - **Minimal TCB** through code partitioning
  - Fewer syscalls need ocalls (instruction to leave the enclave) → **better performance**!
- Cons:
  - **Untrusted memory access** has to be allowed (app data exists outside the enclave)



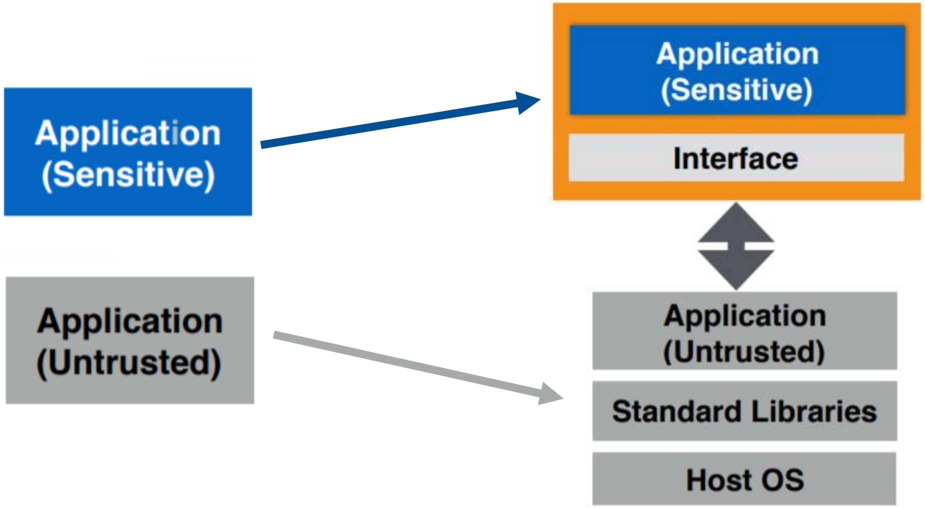
(c) Application-specific interface

What is Glamdring?

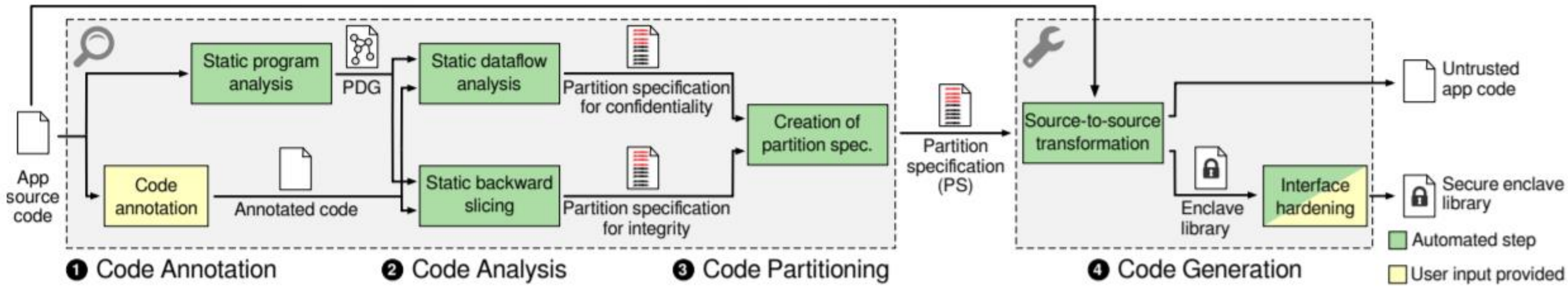
- Glamdring – a framework for protecting existing **C applications** by executing **security-sensitive code** in an **Intel SGX enclave**.

Glamdring: Challenges / Requirements

- Identify security-sensitive code relevant to a security policy (how to determine the minimal TCB?)
- Prevent interfaces from violating security policy
- Avoid performance degradation (enclave crossings?)

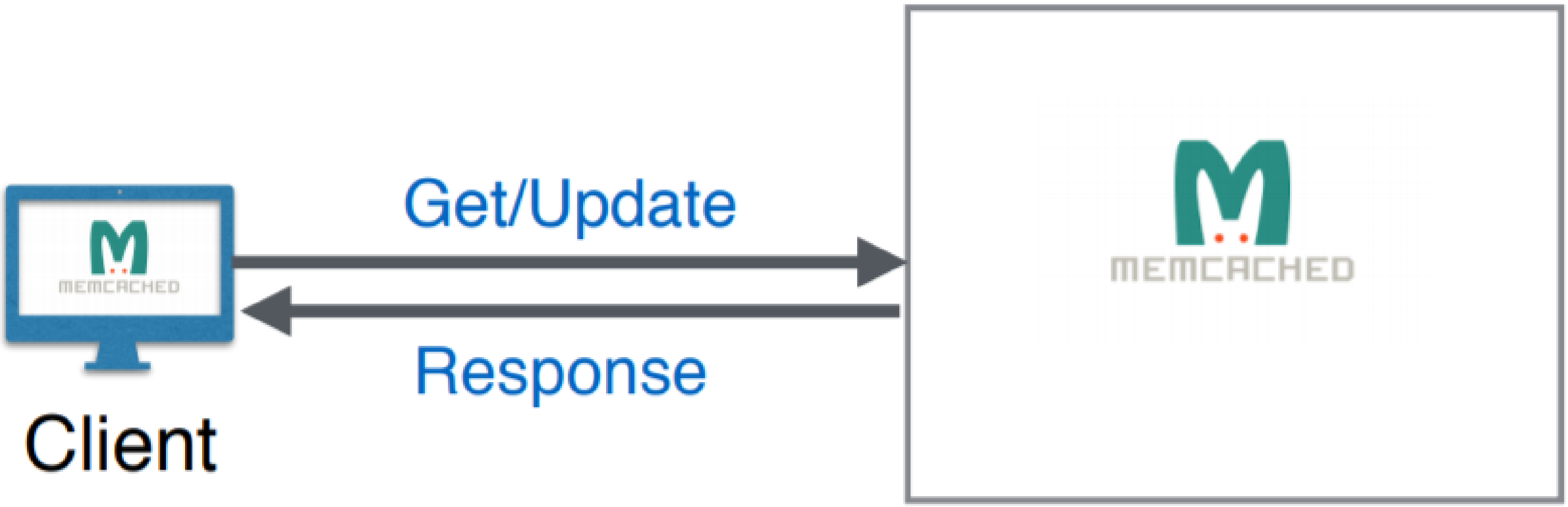


Glamdring Framework Design



Example Application

- Goal: Run Memcached (key-value pair storage) in an enclave
- 2 commands: Get or Update



Code Annotation

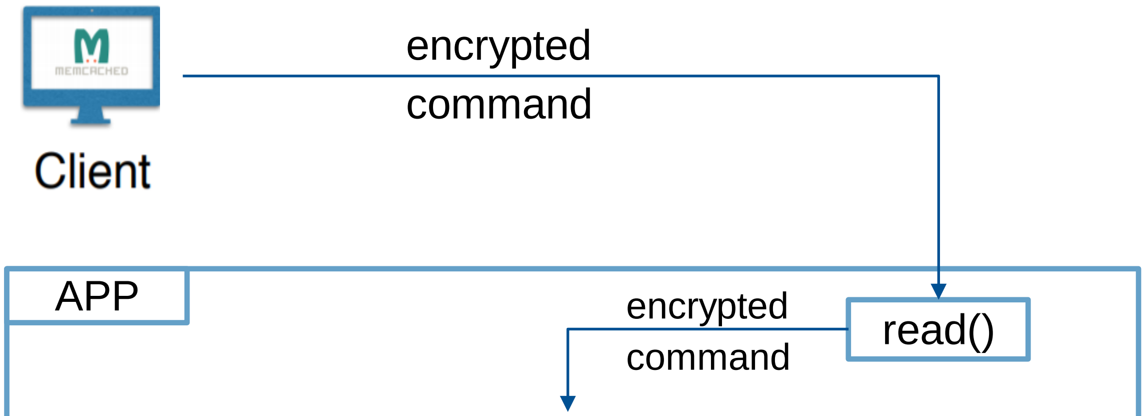
- Glamdring **must** know which application data is sensitive because **sensitive data is application-specific**!
- Developer provides **sources** (inputs) and **sinks** (outputs) of security-sensitive data by **annotating variables** whose values must be protected
- Glamdring relies on the fact that security-sensitive data is **protected** when it is **exchanged** between a **trusted client** and the **application**.

→ Client has to encrypt and sign the data

→ Both the client and the enclave code use symmetric AES-GCM encryption; the key is established upon enclave creation!

Code Annotation: Memcached Example

- Secure-sensitive data – get/update command + request data
- This data is encrypted and signed by the trusted client
- Why we should not annotate socket read() call?

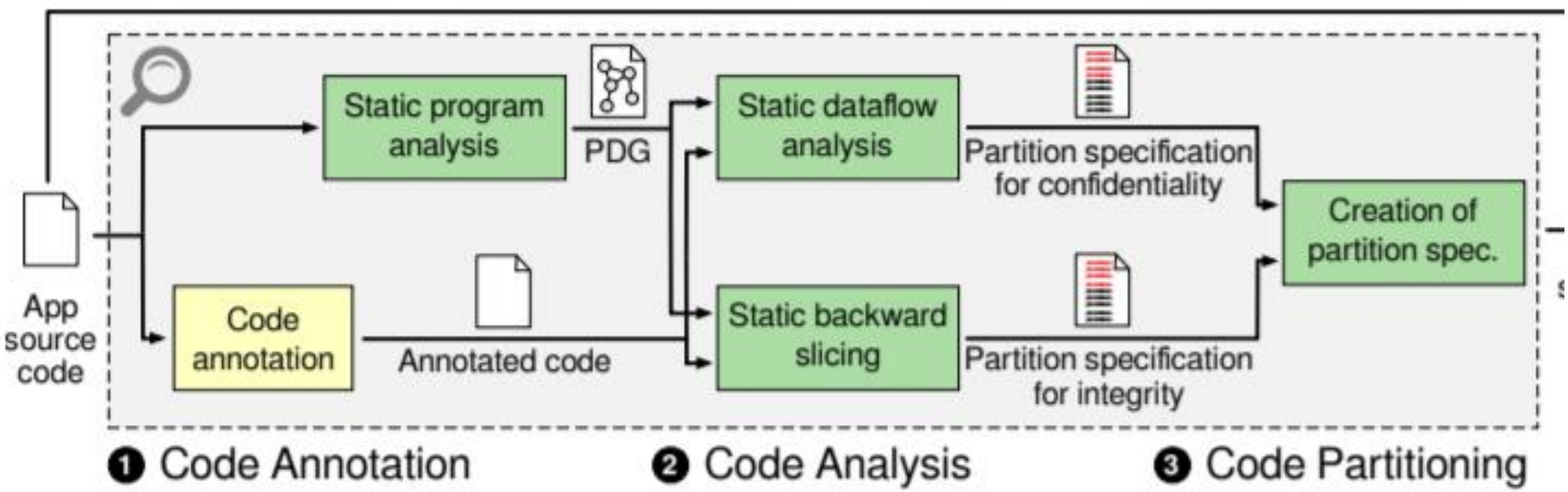



```
#pragma glamdring sensitive-source(command)
static void process_command(conn *c, char *command) {
  token_t tokens[MAX_TOK];
  size_t ntokens;
  ...
  ntokens = tokenize_command(command,tokens,MAX_TOK);
  ...
  process_update_command(c,tokens,ntokens,comm,false);
  ...
}

#pragma glamdring sensitive-sink(buf)
static int add_iov(conn *c, void *buf, int len) {
  ...
  m = &c->msglist[c->msgused - 1];
  m->msg_iov[m->msg_iovlen].iov_base = (void *)buf;
  ...
}
```

Code Analysis

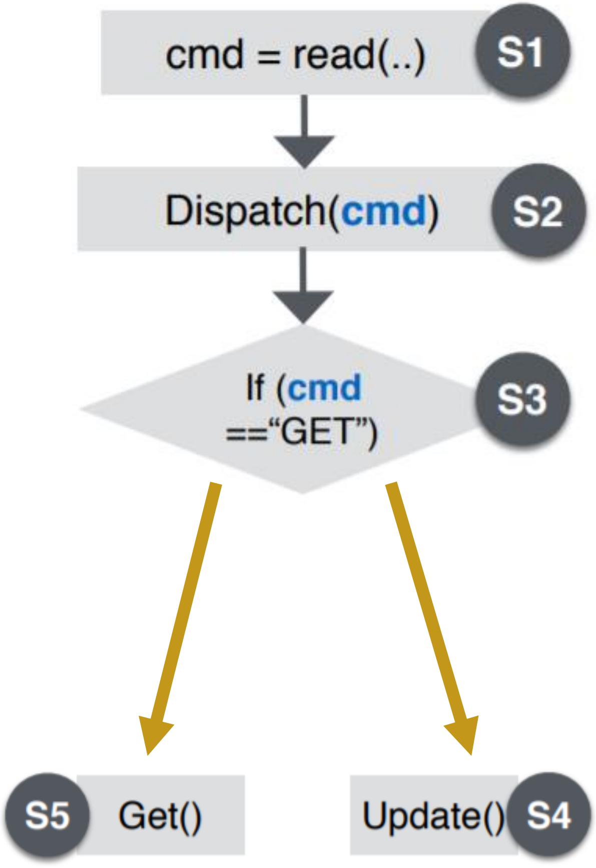
- Goal: Identify all security-sensitive statements in the program that have dependencies on all annotated statements
- Static program analysis: Program Dependence Graph → Static dataflow analysis + Static backward slicing → Partition Specification



Code Analysis: Program Dependence Graph

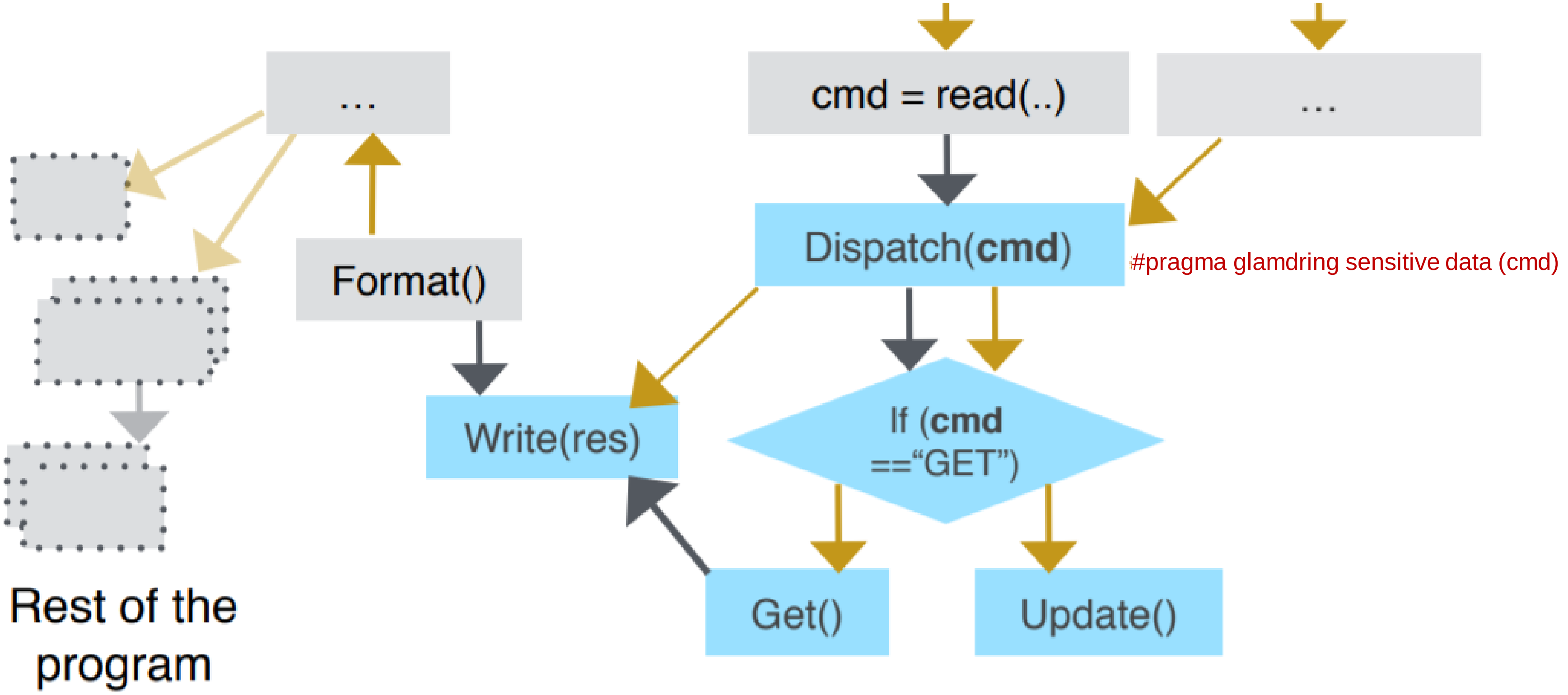
- Captures the **control** and **data** dependencies in the program
- Nodes = Statements = {S1, S2, S3, S4, S5}

- Edges:
  - **Control Dependence Edge**
    - One Statement determines if another gets executed
  - **Data Dependence Edge**
    - Data defined in a statement is used in another statement



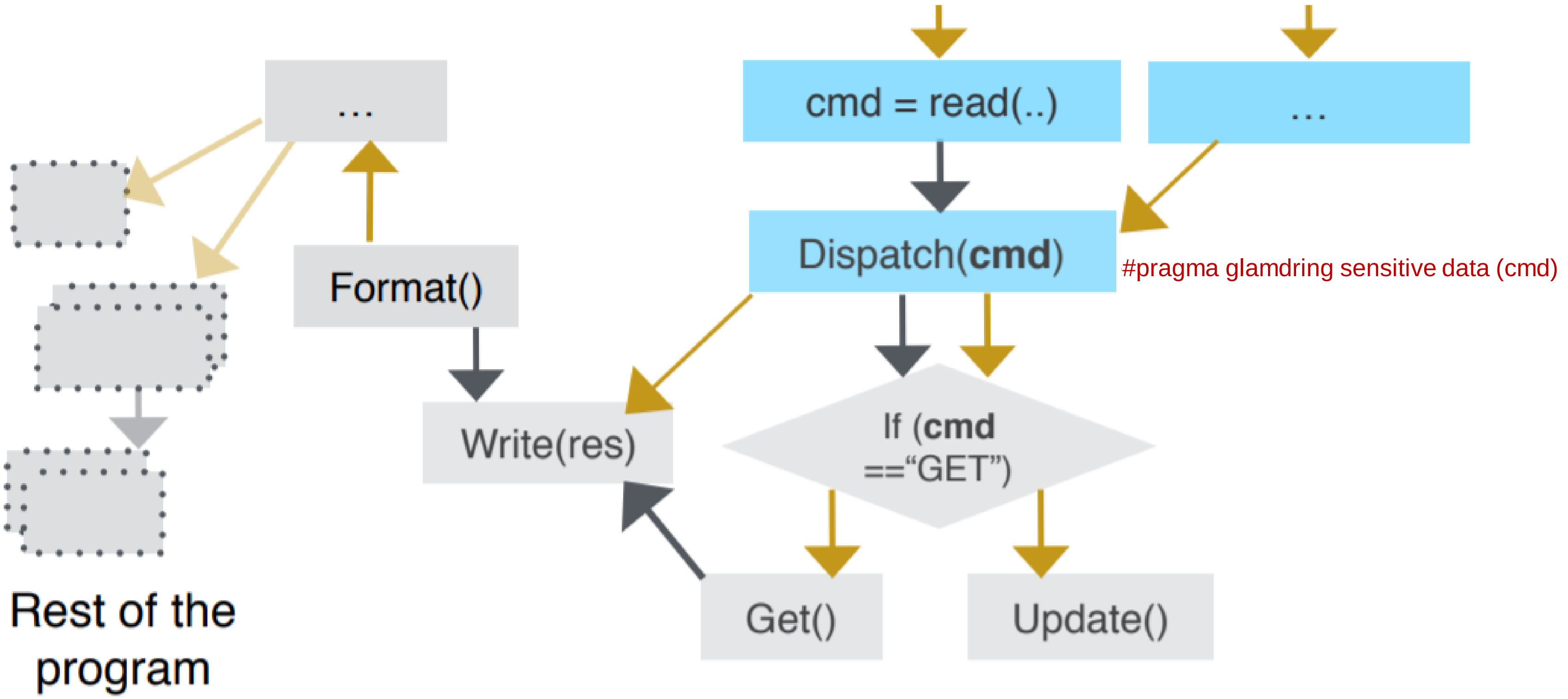
Code Analysis: Static Dataflow Analysis

- **Confidentiality:** Using Graph Reachability identify all nodes which you can reach from annotated node (follow the forward edges)



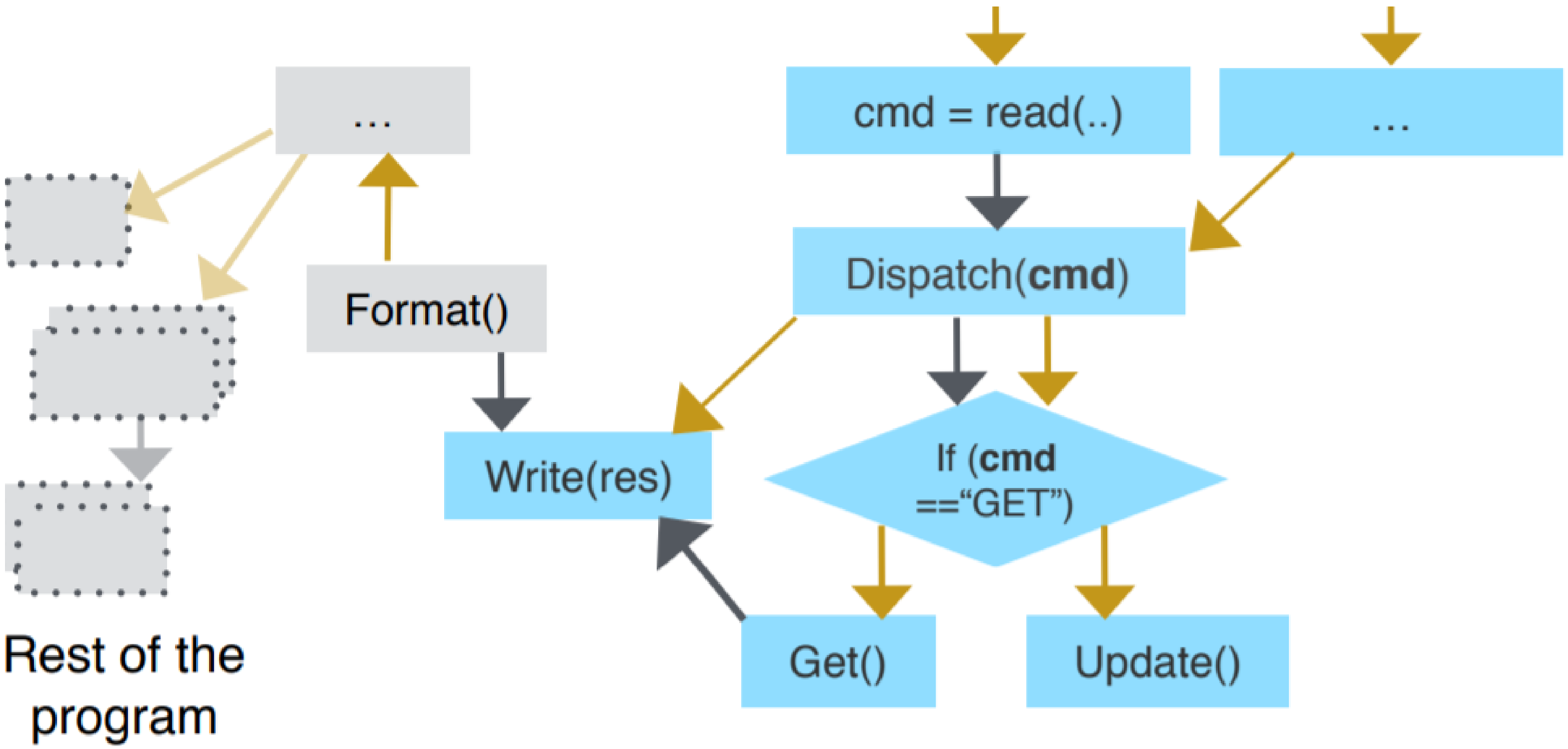
Code Analysis: Static backward slicing

- **Integrity:** Using Graph Reachability identify all nodes which can reach annotated node (follow the back edges)



Code Analysis: Union

- **Union** of nodes found contains the **set of all security-sensitive statements, this set is denoted from now as *S*.**

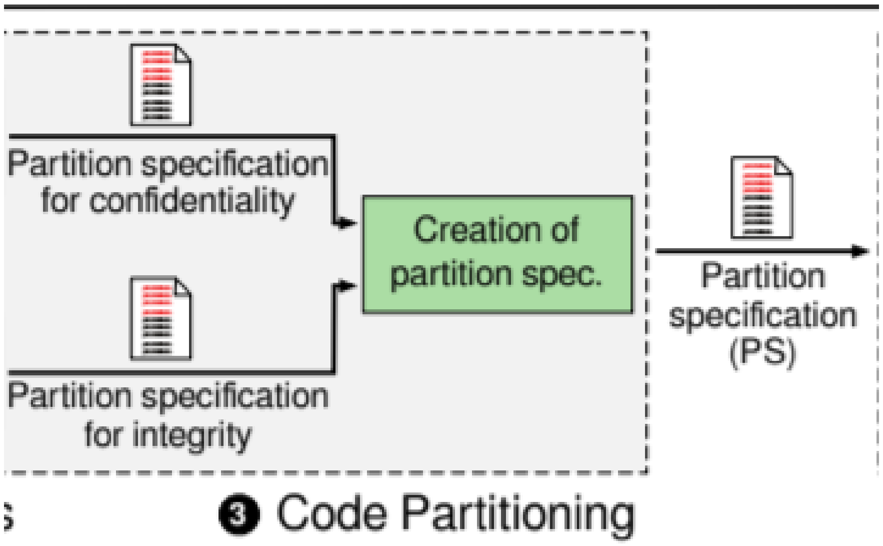


Code Partitioning

- Glamdring produces a partition specification (PS) from the set of security-sensitive statements
- PS contains a set of security-sensitive functions, memory allocations and global variables to protect

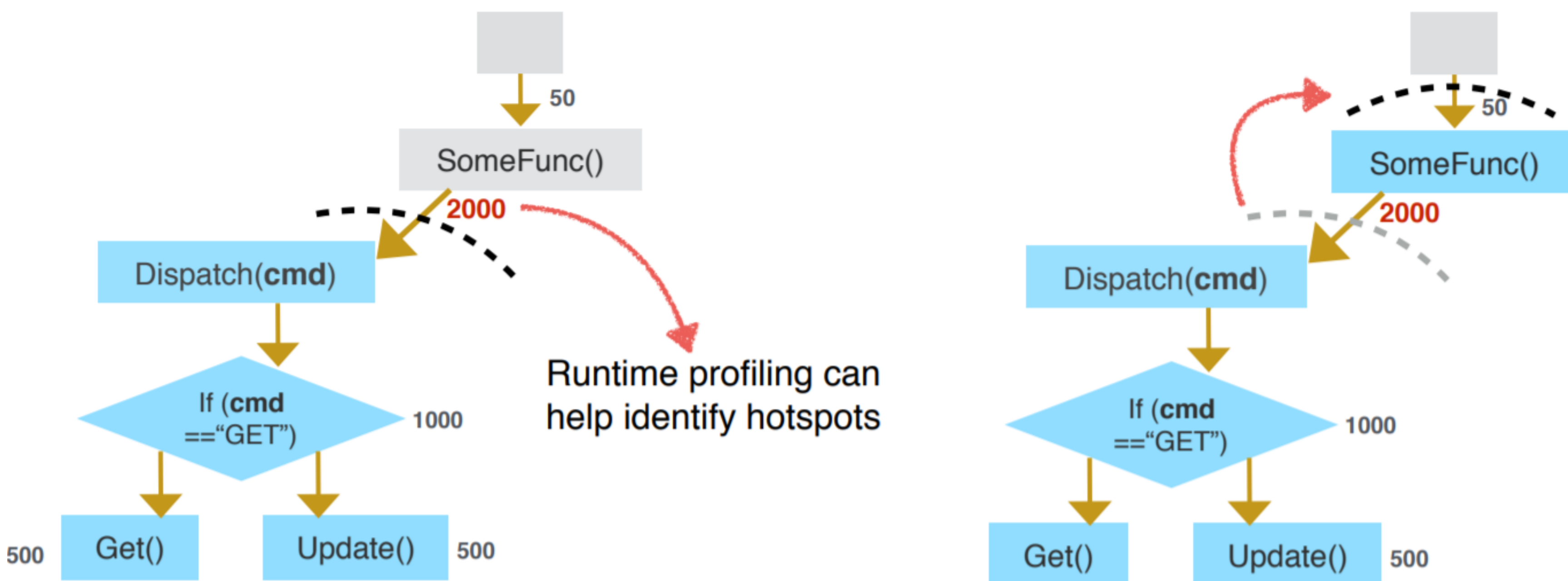
Partition Spec

* Enclave Functions:
  `Dispatch,`
  `Get,`
  `Update`
* Enclave Allocations:
  `malloc@241`
* Enclave Allocated Globals
  `hash_items`



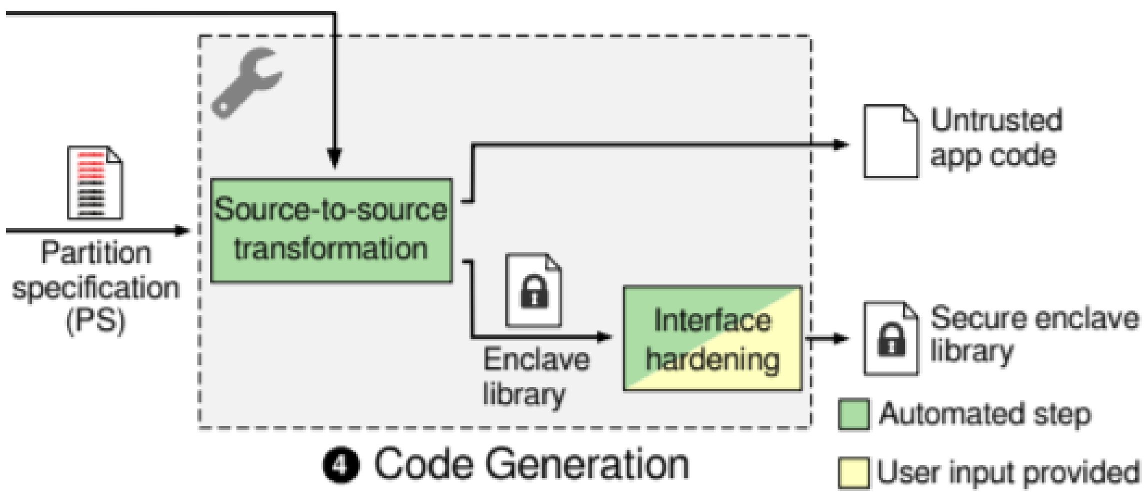
Code Partitioning: Enclave boundary relocation

- Some enclave interface functions may be called too frequently → it results in frequent enclave crossing which reduces performance!
- Solution: configurable threshold, if exceeded Glamdring adds function to the enclave



Code Generation & Hardening

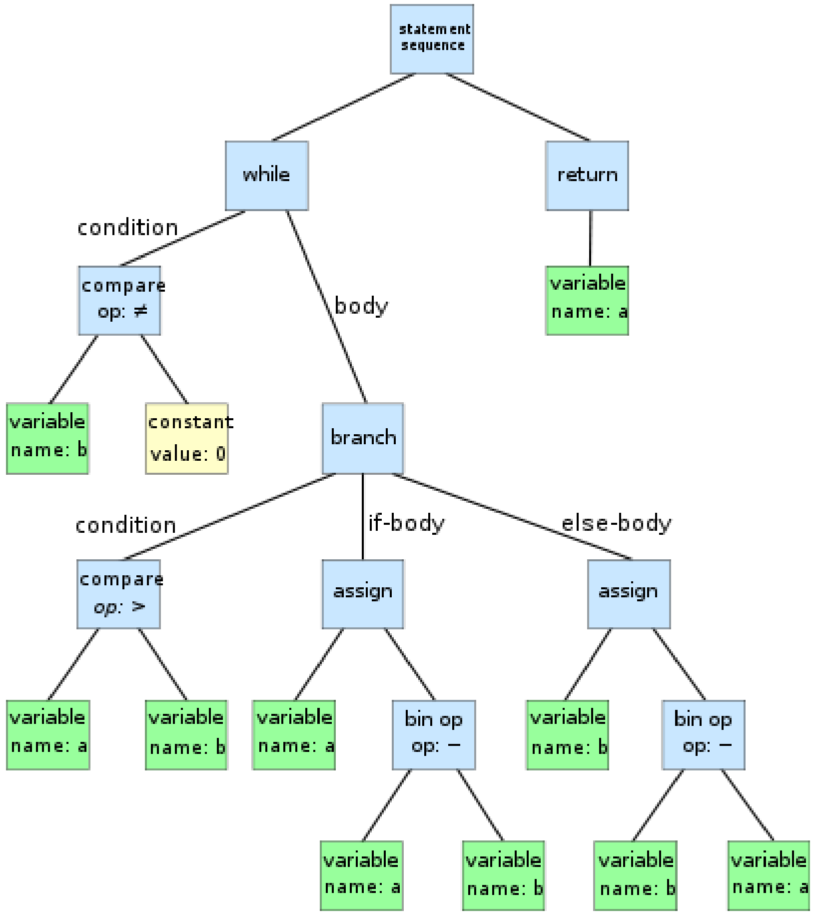
- Produces source-level partitioning of the app based on the PS
- Hardens the enclave boundary against malicious input
- **Result**: Set of enclave and outside source files, along with an enclave specification, which can be compiled using the Intel SGX



Code Generation: Source-to-Source Transform

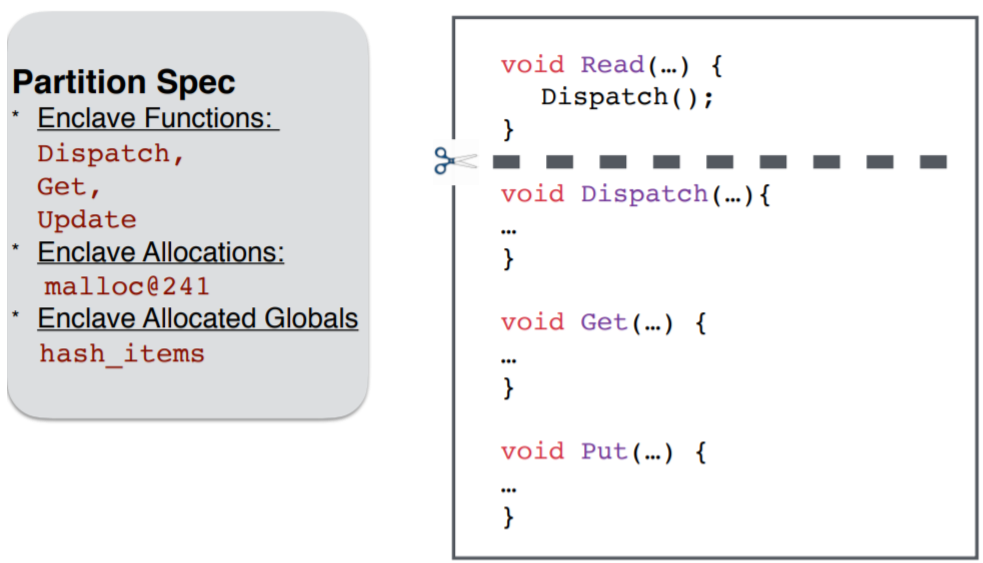
- Relies on the LLVM/Clang compiler toolchain to rewrite the preprocessed C source code → Abstract Syntax Tree

- Code generation from PS 3 step:
  1. Moving function definitions into the enclave
  2. Generating ecalls and ocalls
  3. Handling memory allocation



Code Generation: Moving function definitions

- Code generator creates an enclave version and an outside version for every source file
- Remove all functions not listed in the PS from the enclave version
- Remove all listed enclave functions from the outside version



Code Generation: Generating ocalls and ecalls

- By traversing direct call expressions in each function, code generator identifies the ecalls and ocalls
- If the caller is an untrusted function and the callee is an enclave function → the callee is transformed to an ecall.
- If the caller is an enclave function and the callee is an untrusted function → the callee is transformed to an ocall.

Outside

```
void Read(…) {
   ecall__Dispatch();
}
```

Enclave

```
void ecall__Dispatch(…){
…
}

void Get(…) {
…
}

void Put(…) {
…
}
```

Code Generation: Handling function pointers as interface arguments

- Function pointer arguments to ecalls and ocalls are special cases

```
/* Initialised to func_A and func_B outside */
int (*addrof_func_A)(int); int (*addrof_func_B)(int);

int jump_to_func(int (*fptr)(int), int x) {
  if (fptr==addrof_func_A) return ocall_func_A(x);
  else if (fptr==addrof_func_B) return ocall_func_B(x);
}

int ecall_enclave_func(int (*fptr)(int),int y) {
  return jump_to_func(fptr, y);
}
```

- *Example*: **ecall** passes a function pointer targeting a function on the **outside**, the **program will fail** when the enclave attempts to call that function pointer directly

Code Generation: Handling memory allocation

- Code generator uses PS to decide which mallocs must be placed inside the enclave
- For malloc calls listed in the PS nothing needs to be done because a malloc call inside the enclave allocates memory inside!

- One special case possible:
  - A function must allocate memory outside
  - Arises when placing non-sensitive code into the enclave when:
    - Partitioning at function level
    - Moving functions into the enclave using Enclave Boundary Relocation

- Solution → Malloc is replaced by an ocall to the outside!

Code Hardening

- There is still some attack surface mostly during the code generation phase → protection is needed!

- Possible Attack (infeasible program paths):

```
/* Outside code*/
int dump_flag = 0; // Can be modified by attacker.

/* Enclave code */
int ecall_enclave_func(int dump_flag) {
  char* dump_data = malloc(...);
  if(dump_flag == 1)
    memcpy(dump_data, sensitive_data);
  else
    memcpy(dump_data, declassify(sensitive_data));
  write_to_untrusted(dump_data);
}
```

Code Hardening: Runtime Environment Checks

- To prevent such attacks Glamdring applies runtime checks on global variables and parameters passed into and out of ecalls and ocalls.

- assert(dump_flag == 0) before if statement

```
/* Outside code*/
int dump_flag = 0; // Can be modified by attacker.

/* Enclave code */
int ecall_enclave_func(int dump_flag) {
  char* dump_data = malloc(...);
  if(dump_flag == 1)
    memcpy(dump_data, sensitive_data);
  else
    memcpy(dump_data, declassify(sensitive_data));
  write_to_untrusted(dump_data);
}
```

Evaluation

- Evaluated on 3 different applications:
  - Memcached
  - LibreSSL
  - Digital Bitbox Bitcoin Wallet

- Glamdring Framework Size: 5000 LoC + Static Analysis libraries

Application	Data	Confidentiality	Integrity
Memcached	Key-Value pairs	Yes	Yes
LibreSSL	CA Root certificate	Yes	Yes
Digital Bitbox	Private Keys	Yes	Yes

Evaluation: TCB size

Applications	Code Size (kLoC)	TCB size
Memcached	31	12 (40%)
DigitalBitbox	23	8 (38%)
LibreSSL	176	38 (22%)

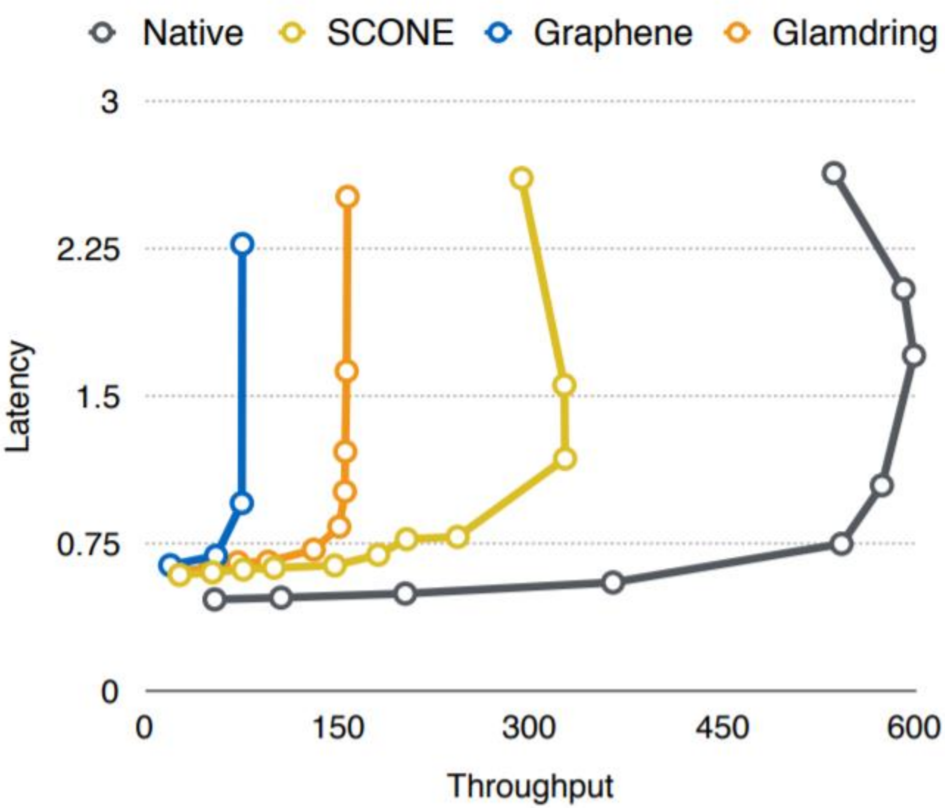
Evaluation: Comparison with Graphene and SCONE

Applications	TCB size (kLoC)	Binary Size
Memcached (Glamdring)	**42**	770 kB
Memcached (SCONE)	149	3.3 MB
Memcached (Graphene)	746	4.1 MB

Evaluation: Performance

- Native: 600k req. per second
- SCONE: 300k req. per second, SCONE does additional optimizations such as user-level threading

- Graphene: 75k req. per second
- Glamdring: 150k req. per second
- Enclave transitions dominate the cost of the request handling → batch requests for better performance (to 200k)



Conclusion

- Glamdring is able to automatically partition the application into trusted and untrusted parts
- This allows us to port untrusted application parts into Intel SGX enclaves
- Which leads to much smaller TCB than prior approaches with acceptable performance