Glamdring: Automatic Application Partitioning for Intel SGX

Joshua Lind
Imperial College London

Christian Priebe
Imperial College London

Divya Muthukumaran
Imperial College London

Dan O’Keeffe
Imperial College London

Pierre-Louis Aublin
Imperial College London

Christian Priebe
Imperial College London

Tobias Reiher
TU Dresden

David Goltzsche
TU Braunschweig

David Eyers
University of Otago

Rüdiger Kapitza
TU Braunschweig

Christof Fetzer
TU Dresden

Peter Pietzuch
Imperial College London

Abstract

Trusted execution support in modern CPUs, as offered by Intel SGX enclaves, can protect applications in untrusted environments. While prior work has shown that legacy applications can run in their entirety inside enclaves, this results in a large trusted computing base (TCB). Instead, we explore an approach in which we partition an application and use an enclave to protect only security-sensitive data and functions, thus obtaining a smaller TCB.

We describe Glamdring, the first source-level partitioning framework that secures applications written in C using Intel SGX. A developer first annotates security-sensitive application data. Glamdring then automatically partitions the application into untrusted and enclave parts: (i) to preserve data confidentiality, Glamdring uses dataflow analysis to identify functions that may be exposed to sensitive data; (ii) for data integrity, it uses backward slicing to identify functions that may affect sensitive data. Glamdring then places security-sensitive functions inside the enclave, and adds runtime checks and cryptographic operations at the enclave boundary to protect it from attack. Our evaluation of Glamdring with the Memcached store, the LibreSSL library, and the Digital Bitbox bitcoin wallet shows that it achieves small TCB sizes and has acceptable performance overheads.

1 Introduction

Applications are increasingly deployed in potentially untrusted third-party data centres and public cloud environments such as Amazon AWS [3] and Microsoft Azure [4]. This has a major impact on application security [1]: applications must protect sensitive data from attackers with privileged access to the hardware or software, such as system administrators. Applications that rely on cryptographic techniques to protect sensitive data [60, 63, 82] limit the operations that can be carried out; fully homomorphic encryption [32] allows arbitrary operations but adds substantial overhead.

A new direction for securing applications in untrusted environments is to use trusted execution mechanisms offered by modern CPUs such as Intel’s Software Guard Extensions (SGX) [42]. With Intel SGX, user code and data are protected as part of secure enclaves. An enclave is a separate memory region that is encrypted transparently by the hardware and isolated from the rest of the system, including higher-privileged system software.

Haven [6], Graphene [55, 81] and SCONE [2] have demonstrated the feasibility of executing entire applications inside enclaves by adding sufficient system support, such as a library OS or the C standard library, to the enclave. By placing all code inside the enclave, these approaches, however, have a large trusted computing base (TCB) that violates the principle of least privilege [67]: all enclave code executes at a privilege level that allows it to access sensitive data. An attacker only needs to exploit one vulnerability in the enclave code to circumvent the security guarantees of trusted execution [78]. The number of bugs even in well-engineered code is proportional to the size of the code [54].

To partially mitigate this problem, proposals for securing applications with enclaves [68, 72, 73] introduce additional checks in enclave code to prevent it from compromising the confidentiality or integrity of enclave data. Such approaches, however, restrict the allowed behaviour of enclave code, e.g. prohibiting general enclave code from interacting with memory outside of the enclave [68]. This limits the applicability of trusted execution mechanisms for arbitrary applications.

We want to explore a different design point for securing applications with trusted execution by placing only security-sensitive functions and data inside the enclave. We exploit the observation that only a subset of all application code is security-sensitive [11, 71, 74], and ask the question: “what is the minimum functionality of an application that must be placed inside an enclave to protect the confidentiality and integrity of its security-sensitive data?” Our goal is to develop a principled approach that (i) partitions applications into security-
sensitive enclave and security-insensitive non-enclave parts; (ii) gives guarantees that the security-sensitive enclave code cannot violate the confidentiality or integrity of sensitive enclave data, even under attack; and (iii) has an acceptable performance overhead despite the limitations of current SGX implementations [16].

In our approach, we use static program analysis to identify a security-sensitive subset of the application code. Being conservative, it allows us to robustly identify the subset of functions that may be exposed to or modify sensitive data. This analysis is independent of application input, which may be controlled by an attacker, and thus is resilient against attacks on the enclave interface, as long as the assumptions made by the static analysis are enforced at runtime.

We describe Glamdring, a new framework for securing C applications using Intel SGX. Glamdring partitions applications at the source code level, minimizing the amount of code placed inside an enclave. To partition an application, a developer first annotates input and output variables in the source code that contain sensitive data and whose confidentiality and integrity should be protected. Glamdring then performs the following steps:

1. **Static dataflow analysis.** To prevent disclosure of sensitive data, functions that may potentially access sensitive data must be placed inside the enclave. Glamdring performs static dataflow analysis [65] to detect all functions that access sensitive data or data derived from it. It tracks the propagation of sensitive data through the application, starting with the annotated inputs.

2. **Static backward slicing.** To prevent an attacker from compromising the integrity of sensitive output data, functions that update sensitive data must be placed inside the enclave. Here Glamdring uses static backward slicing [84], starting from the set of annotated output variables, to identify functions that can affect the integrity of this data. It creates a backward slice with all source code that the sensitive output variables depend on.

3. **Application partitioning.** Glamdring now partitions the application by placing all of the security-sensitive functions identified above inside the enclave. This creates an enclave boundary interface that constitutes all parameters passed to enclave functions and accesses to untrusted global variables. Any sensitive data that crosses the enclave interface is transparently encrypted and signed by the enclave code or trusted remote client, respectively. For performance reasons, some security-insensitive functions may be moved inside the enclave.

4. **Source code generation.** Finally, Glamdring transforms the application using a source-to-source compiler based on the LLVM/Clang compiler toolchain [14, 49]. It (i) generates appropriate entry/exit points at the enclave boundary with the required cryptographic operations; (ii) ensures that memory allocations for data structures are performed inside or outside of the enclave depending on the nature of the data; and (iii) adds runtime checks at the enclave boundary to ensure that the invariants required for the soundness of the static analysis hold. The output of this phase is an untrusted binary and a trusted shared library that executes inside the enclave.

We evaluate the security and performance properties of Glamdring by applying it to three applications: the Memcached key/value store [24], the LibreSSL library [7], and the Digital Bitbox bitcoin wallet [70]. Our experiments show that Glamdring creates partitioned versions of these applications with TCBs that contain 22%–40% of the lines of code of the applications. Despite their strong security guarantees, the partitioned applications execute with between 0.3×–0.8× of the performance of the original versions.

## 2 Background

Protecting application data is crucial. Past incidents have shown that data breaches [41] and integrity violations [75] can have a major impact on users [30] and the reputation of application providers [59].

Today applications are deployed frequently in untrusted environments such as public clouds, controlled by third-party providers. In addition to the application being vulnerable, the underlying infrastructure (i.e. the operating system (OS) and hypervisor) may be untrusted by the application owner, and software-based solutions implemented as part of the OS [17,46] or hypervisor [13,20,39] cannot protect application data.

New hardware security features, such as Intel SGX, offer a solution through a trusted execution model. It supports memory and execution isolation of application code and data from the rest of the environment, including higher-privileged system software. In this work, we address the problem of how developers can protect only the security-sensitive code and data of an application using trusted execution.

### 2.1 Threat model

We consider code to be security-sensitive if it accesses sensitive data directly or can impact the confidentiality or integrity of data indirectly. For example, in the Memcached [24] store, assuming that key/value pairs are sensitive, functions that store key/value pairs are security-sensitive, while ones for network handling are not.

The adversary’s goal is to either disclose confidential data or damage its integrity. We consider a powerful and active adversary, such as a malicious system administrator, who has control over the hardware and software of the machine executing the application. The adversary may therefore (i) access or modify any data in memory or disk; (ii) view or modify the application code; and (iii) modify the OS or other system software.

We do not consider denial-of-service (DoS) attacks—
an adversary with full control over the machine can decide to not run the application. Such attacks can be detected and potentially mitigated using replication [21]. Similar to other work, we also ignore side-channel attacks that exploit timing effects [83] or page faults [86], but there exist dedicated mitigation strategies [10, 19].

2.2 Trusted execution with Intel SGX

Intel’s Software Guard Extensions (SGX) [42] allow applications to protect the confidentiality and integrity of code and data, even when an attacker has control over all software (OS, hypervisor and BIOS) and physical access to the machine, including the memory and system bus.

SGX provides applications with a trusted execution mechanism in the form of secure enclaves. Enclave code and data reside in a region of protected memory called the enclave page cache (EPC). Only application code executing inside the enclave is permitted to access the EPC. Enclave code can access the memory outside the enclave. An on-chip memory encryption engine encrypts and decrypts cache lines in the EPC that are written to and fetched from memory. As enclave code is always executed in user mode, any interaction with the OS through system calls, e.g. for network or disk I/O, must execute outside of the enclave.

Using Intel’s SGX SDK [43], developers can create enclave libraries that are loaded into an enclave and executed by a CPU with SGX support. A developer defines the interface between the enclave code and other, untrusted application code: (i) a call into the enclave is referred to as an enclave entry call (ecall). For each defined ecall, the SDK adds instructions to marshal parameters outside, unmarshal the parameters inside the enclave and execute the function; and (ii) outside calls (ocalls) allow enclave functions to call untrusted functions outside. Added SDK code leaves the enclave, unmashals the parameters, calls the function, and re-enters the enclave.

Any ecalls and ocalls introduce a performance overhead because the hardware must perform certain actions to maintain the security guarantees of SGX. Enclave code must also verify the integrity of accessed data, such as parameters of ecalls, return values of ocalls, and data read from untrusted memory.

2.3 Security with trusted execution

Next we explore the design space for securing application data using trusted execution and discuss the trade-offs with respect to (i) the size of the TCB; (ii) the complexity of the enclave interface; (iii) the development effort; and (iv) the generality of the approach.

With Intel SGX, the TCB consists of the enclave code and the trusted hardware. Following the principle of least privilege [67], only the parts of an application that require access to sensitive data should be executed within an enclave. As studies have shown [54, 69], the number of

![Design alternatives for the use of enclaves](image)

**Figure 1:** Design alternatives for the use of enclaves

---

**Table 1:** Design alternatives for protecting applications using secure enclaves

<table>
<thead>
<tr>
<th>Enclave Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Complete enclave interface</td>
<td>As shown in Fig. 1a, the approach adopted by systems such as Haven [6], SCONE [2] and Graphene [55, 81] provides isolation at a coarse granularity by executing a complete application inside an enclave. Haven runs unmodified Windows applications using the Drawbridge library OS [61]; Graphene uses a library OS in the enclave to run Linux applications; and SCONE places a modified version of the standard C library in the enclave for supporting recompiled Linux applications. Both security-sensitive and insensitive application code and data reside within the enclave, increasing the TCB size.</td>
</tr>
<tr>
<td>(b) Predefined enclave interface</td>
<td>Enclaves are the development effort and whether it is generally applicable to any application.</td>
</tr>
<tr>
<td>(c) Application-specific interface</td>
<td>Important factors that determine the adoption of a given approach for securing applications with secure enclaves are the development effort and whether it is generally applicable to any application. Fig. 1 shows three design alternatives for protecting applications using secure enclaves:</td>
</tr>
</tbody>
</table>

---

**Figure 1:** Design alternatives for the use of enclaves

---

**Table 1:** Design alternatives for protecting applications using secure enclaves

<table>
<thead>
<tr>
<th>Enclave Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Complete enclave interface</td>
<td>As shown in Fig. 1a, the approach adopted by systems such as Haven [6], SCONE [2] and Graphene [55, 81] provides isolation at a coarse granularity by executing a complete application inside an enclave. Haven runs unmodified Windows applications using the Drawbridge library OS [61]; Graphene uses a library OS in the enclave to run Linux applications; and SCONE places a modified version of the standard C library in the enclave for supporting recompiled Linux applications. Both security-sensitive and insensitive application code and data reside within the enclave, increasing the TCB size.</td>
</tr>
<tr>
<td>(b) Predefined enclave interface</td>
<td>Enclaves are the development effort and whether it is generally applicable to any application.</td>
</tr>
<tr>
<td>(c) Application-specific interface</td>
<td>Important factors that determine the adoption of a given approach for securing applications with secure enclaves are the development effort and whether it is generally applicable to any application. Fig. 1 shows three design alternatives for protecting applications using secure enclaves:</td>
</tr>
</tbody>
</table>
as it can execute mostly unmodified applications, and is
generic across applications, it cannot mask fundamental
limitations of SGX when trying to provide a complete
enclave interface. For example, SCONE cannot support
applications that use the fork() system call.

**Predefined enclave interface.** Fig. 1b shows an
approach in which applications must adhere to a predefined restricted enclave interface [68, 72, 73]. For example, VC3 [68] protects map/reduce jobs using enclaves and forces map/reduce tasks to interact with the untrusted
environment only through a particular interface. The enclave contains a small trusted shim library, resulting in a
smaller TCB compared to the previous approach.

This approach results in a minimal enclave interface—
VC3’s interface consists of only two calls, one to read
encrypted key/value pairs and another to write them as
the job output. This limited interaction of the enclave
with the outside simplifies protection: it is possible to
add dynamic checks that enforce security invariants [72],
e.g. preventing enclave code from accessing untrusted
memory except through the enclave interface.

The security benefits of this approach are offset by its
limited applicability. Given the predefined enclave inter-
face, the approach can only be used with applications that
interact with the untrusted environment in specific ways,
such as map/reduce tasks.

**Application-specific enclave interface.** We explore an-
other design point. We exploit the fact that, for many ap-
lications, only a subset of code handles sensitive data,
while other code is not security-sensitive and does not
need protection [9, 71, 74]. As shown in Fig. 1c, this makes it possible to **partition** the application to reduce the
TCB size, leaving code and data that is not security-
sensitive outside the enclave.

Past work has shown that partitioning can be done by
hand so that complex applications can exploit enclaves [9,
58]. Instead, we want to explore the hypothesis that it is
feasible to use principled techniques, such as program
analysis, to partition applications for secure enclaves, and
provide security guarantees about the enclave code and
its interface to the untrusted environment.

With this approach, the enclave interface now becomes
**application-specific**: a set of ecalls and ocalls is required
between trusted and untrusted application code. In con-
trast to a complete enclave interface, fewer system calls
need ocalls because application code that is placed out-
side the enclave can issue system calls directly.

Since application data now also exists outside the en-
clave, enclave code must be allowed to access untrusted
memory. This means that it is no longer possible to pro-
hibit all memory accesses, as with the predefined enclave
interface [72]. Instead, it is important to give security
guarantees that, despite the richer application-specific en-
clave interface, the untrusted environment cannot disclose
sensitive enclave data or compromise its integrity.

## 3 Glamdring Design

We present **Glamdring**, a framework for protecting exist-
ing C applications by executing security-sensitive code in
an Intel SGX enclave. Glamdring targets the following re-
quirements: it must protect the confidentiality of sensitive
input data and the integrity of sensitive output data (R1);
apply the principle of least privilege, minimising the code
that can access sensitive data (R2); automate changes to the
application code (R3); and impose an acceptable perfor-
mance overhead (R4). To achieve these requirements,
Glamdring operates in four phases (see Fig. 2):

1. **Code annotation**: Glamdring must know which ap-
lication data is sensitive (R1). The developer provides
information about the sources (inputs) and sinks (outputs)
of sensitive data by annotating variables whose values
must be protected in terms of confidentiality and
integrity (§3.1).

2. **Code analysis**: Based on the annotated source code,
Glamdring identifies a subset of code that is security-
sensitive (R2). It uses automatic static program analy-
sis (R3) to find control and data dependencies on security-
sensitive data. Glamdring thus obtains the minimal set of
statements that either handle confidential data or affects
its integrity (R1/R2) (§3.2).

3. **Code partitioning**: Next Glamdring creates a partition specification (PS) that defines which parts of the code
must be protected by the enclave. The PS enumerates the
functions, memory allocations and global variables that
are security-sensitive based on the program analysis. This
defines the enclave boundary interface of the partitioned
application, which includes ecalls, ocalls, and direct ac-
tesses to untrusted memory (§3.3).

4. **Code generation**: Finally, Glamdring uses a source-
to-source compiler that, based on the PS, partitions the
code into a secure enclave library and untrusted code.
The enclave boundary interface is hardened with runtime
checks that enforce invariants on the program state (§4).

3.1 Code annotation phase

The security-sensitive data that must be protected is application-dependent. To identify it, a developer must therefore annotate the source code. Glamdring relies on the fact that security-sensitive data is protected when it is exchanged between a trusted client and the application. For example, if data is received via the network, Glamdring requires the client to encrypt and sign the data. Both the client and the enclave code use symmetric AES-GCM encryption [22]; the key is established upon enclave creation.

When encrypted security-sensitive data reaches the application through a source, such as an I/O channel, or leaves the application through a sink, a developer must annotate the corresponding variable using a compiler pragma. The annotation sensitive-source identifies a variable at a given source code location where security-sensitive data enters the application; analogously, the annotation sensitive-sink indicates a variable at which security-sensitive data leaves the application.

For example, for Memcached we assume that the security-sensitive data is the type of command submitted by the client (get/set) and its associated key/value data. This data is encrypted and signed by the trusted client when sent to the application. Using Glamdring, Memcached then requires two annotations:

```
#pragma glamdring sensitive-source(command)
static void process_command(conn *c, char *command) {
  token_t tokens[MAX_TOK];
  ...
  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;
}
```  

An obvious location for the sensitive-source annotation might be the socket read() call from which a client request is received. However, this would be unnecessarily conservative because it would denote all network data as security-sensitive (and thus encrypted). Instead, the annotation in line 1 marks the content of the parameter command, which holds the request command and data, as security-sensitive. The sensitive-sink annotation in line 12 specifies that the output buffer for the client response also contains security-sensitive data.

3.2 Code analysis phase

Next the code analysis phase identifies all security-sensitive statements in the program that have dependencies on the set of all annotated statements $S_A$. This combines (a) for confidentiality, the set of all statements that are influenced by the ones in $S_A$; and (b) for integrity, the set of all statements that influence the ones in $S_A$.

Glamdring uses static program analysis to identify all security-sensitive statements. Static analysis is workload-independent and hence makes conservative decisions about dependencies. To ensure that an attacker cannot violate the invariants that static analysis infers from the untrusted code, Glamdring adds runtime checks during code generation (see §4).

Glamdring’s analysis uses a program dependence graph (PDG) [23], referred to as P, in which vertices represent statements, and edges are both data and control dependencies between statements. PDGs are effective representations for program slicing [40, 56]. Using P, Glamdring finds the set of all security-sensitive statements as follows:

1. **Static dataflow analysis for confidentiality.** Given $S_A$ and $P$, Glamdring uses graph-reachability to find a subgraph $P_s$ of $P$ that contains all statements with a transitive control/data dependence on statements in $S_A$ (i.e., vertices reachable from statements in $S_A$ via edges in $P$).

For statements in $S_A$ that are annotated as a sensitive-sink, Glamdring encrypts/signs the data before the statement inside the enclave, making it unnecessary to perform dataflow analysis from these statements.

2. **Static backward slicing for integrity.** Given $S_A$ and $P$, Glamdring uses static backward slicing to find a subgraph $P_i$ with all statements in $P$ on which statements in $S_A$ have a control/data dependence (i.e., all vertices from which statements in $S_A$ are reachable via $P$).

For these statements in $S_A$ that are annotated as sensitive-source, Glamdring employs client-side encryption of the data, making it unnecessary to perform backwards slicing from these statements.

Finally, the set of all security-sensitive statements $S_s$ is obtained by combining $P_s$ and $P_i$.

3.3 Code partitioning phase

Although $S_s$ enumerates security-sensitive statements, Glamdring partitions the application at the granularity of functions rather than statements. This makes the enclave boundary coincide with the application’s function interface, easing automatic code generation (§4) and minimising the required code changes (R3).

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

(i) **functions**: the PS includes all functions whose definitions contain at least one statement in $S_s$;

(ii) **memory allocations**: the PS must identify allocated memory for security-sensitive data. Statements in $S_s$ with calls to malloc (or similar) are enumerated in the
PS, and these allocations are placed inside the enclave;
(iii) global variables: the PS lists all global variables
accessed in statements in $S_e$ and these are allocated inside the enclave. Special accessor ecalls (with checks) are
provided to the untrusted code to access these globals if needed. The PS specifies if the global was part of $P_c$ or
$P_f$ or both, which determines what type of access (read,
write or none) the outside code has.

Enclave boundary relocation (EBR). Glamdring’s code analysis phase produces a lower bound on the code that
must be inside the enclave to guarantee security. In practice, however, a partitioning may prove costly in terms of performance if program execution must frequently cross the enclave boundary interface. Glamdring improves performance by moving additional functions into the enclave in order to reduce the number of enclave crossings. Using a representative workload and the output of the gcov runtime profiling tool [28], Glamdring assigns a cost to each enclave boundary function according to the number of invocations. Up to a configurable threshold, Glamdring adds functions to the enclave. Adding extra functions to the enclave cannot violate the security guarantees of Glamdring, but it does increase the TCB size.

3.4 Discussion
The security guarantees of Glamdring rely on (a) the soundness of the static analysis; (b) the modeling of external library calls whose source code is unavailable; and (c) the correctness of annotations.

Static analysis. To be tractable, static analysis infers invariants on program state based on the source code. These invariants must also hold at runtime, even when the untrusted code is under control of an attacker. As we describe in §4.2, Glamdring ensures this by adding runtime invariant checks to the enclave boundary.

Static pointer analysis is undecidable for C programs [64] and thus fundamentally imprecise [33, 38]. The existence of false positives, however, does not compromise soundness: the partitioning phase may assign more functions to the enclave than necessary, but never excludes security-sensitive functions from the enclave.

Modelling external library calls. Static analyses must model the behaviour of all invoked functions, including those in external libraries with unavailable definitions. A conservative model makes all output parameters dependent on all input parameters and hence upholds the security guarantees; more precise models can consider actual function behaviour to specify dependencies [5, 36].

Annotations. Most static analysis tools for security rely on developer annotations of sources/sinks of security-sensitive data [35, 76]. While these are application-specific, in many cases they are easy to identify, e.g. when they are well-known library functions for I/O channels.

4 Code Generation and Hardening
The code generation phase produces a source-level partitioning of the application based on the partition specification (PS) (§4.1). In addition, it hardens the enclave boundary against malicious input, ensuring that the enclave upholds the confidentiality and integrity guarantees for sensitive data (§4.2). The result is a set of enclave and outside source files, along with an enclave specification, which can be compiled using the Intel SGX SDK.

4.1 Code transformation
The code transformation must (a) handle calls into and out of the enclave; and (b) change the allocation, scope and lifetime of variables and functions in the generated enclave and non-enclave versions of the code.

Glamdring provides a code generator that relies on the LLVM/Clang compiler toolchain [14, 49] to rewrite the preprocessed C source code. It uses the Clang libraries to parse source code into an abstract syntax tree (AST), and traverses the AST to analyse and modify the source code. In addition to the enclave and outside source files, it produces an interface specification in the enclave definition language (EDL) required by the Intel SDK [43]. The code generation proceeds in three steps:

(i) Moving function definitions into the enclave. For each source file, the code generator creates an enclave and an outside version, which contain a copy of the original preprocessed C source file. From the enclave version, it removes all functions not listed in the PS; from the outside version, it removes all listed enclave functions.

(ii) Generating ecalls and ocalls. Based on the set of enclave functions, the code generator identifies the ecalls and ocalls that are part of the enclave boundary interface. It traverses the direct call expressions in each function:
(a) if the caller is an untrusted function and the callee is an enclave function, the callee is made an ecall; (b) if the caller is an enclave function and the callee is an untrusted function, the callee is made an ocall.1

Adding stubs for encryption/decryption. As mentioned in §3.1, the security-sensitive data received from (and returned to) clients is encrypted (and integrity-protected) using a shared AES-GCM key. The code generator adds code to (a) decrypt security-sensitive data entering the enclave at locations annotated as sensitive-source, and (b) encrypt the security-sensitive data leaving the enclave at locations annotated as sensitive-sink. The application client must be modified to handle the corresponding encryption/decryption operations.

Handling C library functions. Calls to C library functions are handled separately. A subset is supported by the Intel SDK inside the enclave and is handled in a polymor-

1Pointers passed outside the enclave are only deep-copied if data in enclave-allocated memory needs to be declassified—the programmer needs to implement this manually.
The enclave code is, by construction, immune to input manipulation attacks. As long as Glamdring’s static analysis is sound, it transitively identifies all code that can affect the confidentiality and integrity of security-sensitive data annotated by the developer, placing it inside the enclave (see §3.2).

However, static analysis infers invariants about the possible values of program variables at different program points, permitting it to prune unfeasible program paths from analysis. The soundness of the static analysis therefore depends on these invariants holding at runtime. Any invariant that relates to untrusted code or data may be compromised by an attacker. The following code snippet gives an example of a debug option that is deactivated in the source code:

```c
/* 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);
}
```

Static analysis infers that the value of dump_flag cannot be 1, making it impossible to take the branch that does not include the declassify() call. Since the value of dump_flag does not affect the control flow leading to sensitive data release, Glamdring would allocate it outside the enclave. An attacker could set dump_flag to any value at runtime, including 1, to cause data disclosure.

Runtime invariant checks. To prevent such attacks, Glamdring enforces the invariants assumed by the static analysis at runtime. It does this by extracting invariants from the analysis phase and adding them as runtime checks in the code generation phase. Glamdring applies checks on global variables and parameters passed into and out of ecalls and ocalls. In the above example, Glamdring adds a check assert(dump_flag == 0).

Checks are also applied to pointers. The static analysis infers the subset of malloc calls that may allocate memory pointed to by each pointer. Glamdring distinguishes between two cases: (a) the analysis infers that a pointer may only point to untrusted memory. A runtime check upholds this and any other invariants on pointer aliasing; or (b) the pointer may point to enclave memory. Here, Glamdring’s invariant checks prevent pointer-swapping attacks (i.e. a trusted pointer being replaced by another trusted pointer): Glamdring instruments the malloc calls inferred for that pointer inside the enclave, storing the addresses and sizes of allocated memory. When a trusted pointer is passed to the enclave via an ecall, it is checked to ensure that it points to a memory region allocated by one of the statically inferred malloc calls for that pointer.

This upholds the results of the static pointer analysis at runtime with enclave checks.

Handling function pointers as interface arguments. Function pointer arguments to ecalls and ocalls are special cases because the target function may not exist at the point of invocation of the function pointer. For example, if an 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. Glamdring employs a static function pointer analysis [89] to identify the possible target functions of function pointer arguments passed to ecalls and ocalls. The code generator then creates ecalls or ocalls for the target functions and uses a trampoline to jump to the correct one, as shown in the jump_to_func function:

```c
int jump_to_func(int (*fptr)(int), int x) {
    int (*addrof_func_A)(int); int (*addrof_func_B)(int);
    /* Initialised to func_A and func_B outside */
    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);
    }
}
```

(iii) Handling memory allocation. The code generator also uses the PS to decide which memory allocations must be placed inside the enclave. For the memory allocations listed in the PS, nothing needs to be done because a malloc call inside the enclave allocates memory inside; for other memory allocations, a function must allocate memory outside, and the malloc is replaced by an ocall to the outside. This arises when placing non-sensitive code into the enclave when (i) partitioning at the function instead of statement level; and (ii) moving functions into the enclave using EBR (see §3.3).

4.2 Code hardening

Next we analyse the attack surface of the enclave boundary interface and describe the protection techniques of the code generation phase against attacks (R1).

Interface attacks. The security of the enclave code depends on the inputs that it receives from the enclave interface. An attacker may manipulate the parameters to ecalls, the results of ocalls, and accesses to globals.

Secure by construction: The enclave code is, by construction, immune to input manipulation attacks. As long as

2Linked calls to the few stateful C library functions (e.g. strtok) typically do not span multiple functions, making it unlikely that such calls get partitioned into different regions.
For checks on global variables allocated outside, before each use, Glamdring copies the value inside and applies the check to the local copy.

**Enclave call ordering attacks.** By construction, Glamdring prevents an attacker from subverting the security guarantees by changing the order in which ecalls are invoked. The transitivity of static analysis ensures that all functions that have a data/control flow dependence relationship (in either direction) with security-sensitive data are placed inside the enclave. Therefore, any change in the ordering of ecalls cannot affect the security guarantees as long as the statically-inferred enclave boundary is enforced. The EBR operation does change this boundary, but only by placing extra functions inside, and therefore cannot violate the security guarantees.

**Iago attacks.** For applications that use C library functions unavailable in Intel SGX SDK, Glamdring adds ocalls (see §4.1). The arguments to such ocalls may expose security-sensitive data or their results may cause integrity violations, leading to Iago attacks [12]. For these functions, Glamdring enforces statically inferred invariants on the return values at runtime. Further protection could be done similar to I/O shields in SCONET [2].

**Replay attacks.** An attacker may tamper with the program state assumed by the enclave by replaying previously issued ecalls. Glamdring guarantees the freshness of encrypted sensitive data that is passed to ecalls. The client affixes a freshness counter to security-sensitive data as part of its encryption (see §3.3). The enclave stores the latest freshness counter for each data item, and validates freshness at ecalls. After an enclave restart, the freshness counters must be restored to their latest values [77].

**Enclave code vulnerabilities.** Enclave code may contain vulnerabilities that can be exploited by an attacker. By reducing the amount of code executed in the enclave, Glamdring makes it more feasible to apply existing techniques to discover and rectify bugs such as buffer overflows [37,48], data races [45] and memory leaks [47].

5 Evaluation

We evaluate Glamdring by applying it to the Memcached key/value store [24], the LibreSSL library [7] and the Digital Bitbox bitcoin wallet [70]. §5.1 describes the security objectives, the source code annotations and the resulting partitioning and its interface. The TCB (LOC) identified by Glamdring varies between 22% and 40%, and the size of the interface between 41–171 ecalls and 51–615 ocalls for the three applications. §5.2 presents performance results on SGX hardware: the partitioned applications execute with 0.5×–0.8× of the native performance.

**Glamdring implementation.** Glamdring uses the Frama-C Aluminium [25] static analysis framework, with the “Impact Analysis” [26] and “Slicing” [27] plug-ins and CodeSurfer 3.0.0 [34]. The Glamdring code generator uses LLVM/Clang 3.9 and has approx. 5,000 LOC.

**Memcached** [24] is a distributed key/value store. It supports several operations: set(k, v), get(k), delete(k), and increment/decrement(k, i). We apply Glamdring to Memcached 1.4.25 that includes lievent 1.4.14 [62], an asynchronous event library. Memcached has 31,100 LOC and 655 functions.

**LibreSSL** [7] is a fork of the OpenSSL cryptographic library [18], with the goal to provide a simpler and more secure implementation. We apply Glamdring to LibreSSL 2.4.2 to secure its functionality when serving as a certificate authority (CA). LibreSSL has 176,600 LOC and 5,508 functions, which are divided into three libraries, libcrypto, libssl and apps/libopenssl. We compile LibreSSL without inline assembly because our static analysis does not support it.

**Digital Bitbox** [70] is a bitcoin wallet designed for high-security USB microcontrollers. It supports: (i) hierarchical deterministic key generation; (ii) transaction signing; and (iii) encrypted communication. We apply Glamdring to Digital Bitbox 2.0.0 with Secp256k1 1.0.0, a cryptographic library, and Yajl 2.1.0, a JSON library. Digital Bitbox has 23,300 LOC and 873 functions.

5.1 Security evaluation

We evaluate the security of the partitioned application in terms of the TCB size and the exposed enclave interface.

5.1.1 Memcached

**Security objectives.** We want to protect the integrity and confidentiality of all key/value pairs in an untrusted Memcached deployment, preventing an attacker from reading or modifying the stored key/value data. For this, we use the source code annotations described in §3.1.

**Security-sensitive code.** Tab. 1 shows that Glamdring places 40% of LOC, 42% of functions and 68% of global variables of Memcached inside the enclave. EBR moves a single additional function into the enclave, reducing the ocall crossings by an order of magnitude for get and set operations. We conclude that a large portion of the Memcached codebase (without lievent) is security-sensitive, as 87% of its functions and 85% of its global variables are assigned to the enclave.

**Partitioned architecture.** Glamdring places the following Memcached functionality inside the enclave: (i) binary/ASCII protocol handling functions; (ii) slab and cache memory management functions that manipulate the data structures responsible for the internal storage of key/value pairs; and (iii) the hash functions over key/value pairs. The functionality placed outside includes: (i) thread initialization and registration functions; (ii) lievent functions for socket polling and network I/O; and (iii) signal handlers and string utility functions.

**Enclave interface.** The enclave interface (see Tab. 1) has 41 ecalls and 146 ocalls. Out of these, 82 ocalls are to C

![Image](image.png)
library functions unavailable inside the enclave; 64 \texttt{ocalls} are to application functions.

To protect the security-sensitive data between the Memcached client and the enclave interface, Glamdring encrypts the following parameters at the client for each request: (i) the operation to perform; (ii) the key; and (iii) the value. The keys, values and the request outcome are encrypted in the client response.

### 5.1.2 LibreSSL

#### Security objectives.
Our goal is to protect the confidentiality of the private key of the root certificate of the LibreSSL CA. We annotate the private key as follows:

```c
int ca_main(int argc, char ** argv) {
    ...
    #pragma glamdring sensitive-source(pkey)
    pkey = load_key(bio, keyfile, keyform, 0, key, "...");
    ...
}
```

#### Security-sensitive code.
Tab. 1 shows that Glamdring places 22\% of LOC, 17\% of functions and 16\% of global variables inside the enclave. EBR moves 2 functions into the enclave, thereby: (i) more than halving the number of \texttt{ecall} crossings; and (ii) reducing the number of \texttt{ocall} crossings by an order of magnitude for sign requests. The majority of functions and global variables assigned to the enclave originate from the libcryptio library, which contains most of the certificate signing logic.

#### Partitioned architecture.
Glamdring places only a subset of LibreSSL into the enclave: (i) the entropy/random number generator; (ii) the RSA and Big Numbers module; and (iii) the X509 module, which stores the certificates. The functionality placed outside includes: (i) the TLS/SSL modules for secure communication; (ii) digest algorithms (MD5, SHA256); and (iii) cryptographic protocols unrelated to certificate signing (DSA, AES)

#### Enclave interface.
LibreSSL exposes 171 \texttt{ecalls} and 613 \texttt{ocalls} (see Tab. 1). Out of those, only 23 \texttt{ocalls} provide access to C library functions; 49\% of \texttt{ocalls} provide access to global variables; and the remaining 278 \texttt{ocalls} are used to execute outside LibreSSL functions.

Glamdring places the private key of the root certificate and any variables that depend on it inside the enclave. The communication between the client requesting a certificate signature and the enclave involves: (i) reading the certificate to be signed; and (ii) outputting the signature. We assume that the root certificate and its private key are given to the enclave during initialisation [44]. Since the signed certificate is not confidential, no explicit declassification is needed before writing it to disk via an \texttt{ocall}.

### 5.1.3 Digital Bitbox

#### Security objectives.
We want to secure Digital Bitbox in a remote deployment, such as an online bitcoin service. An attacker must not (i) read/modify the private keys in the wallet; and (ii) issue commands such as transactions.

We consider three API calls security-sensitive: (i) \texttt{seed()} to create a new wallet; (ii) \texttt{sign()} to sign a transaction and return the signature; and (iii) \texttt{random()} to return a random number. We annotate these API calls with security annotations. The listing below shows the annotation added to protect the transaction signature returned to the user for the \texttt{seed()} API call:

```c
int wallet_sign(char *message, char *keypath) {
    uint8_t sig[64];
    ...
    ecc_sign_digest(node.private_key, data, sig)
    ...
    #pragma glamdring sensitive-source(sig)
    return commander_fill_signature_array(sig, pub_key);
}
```

#### Security-sensitive code.
Glamdring places 38\% of LOC, 42\% of functions and 52\% of global variables inside the enclave (see Tab. 1). EBR increases the TCB by 4 functions, reducing the number of \texttt{ecall} and \texttt{ocall} crossings at runtime by between 1 and 3 orders of magnitude, for the \texttt{seed}, \texttt{sign} and \texttt{random} API calls. Only half of the Digital Bitbox code itself is security-sensitive: 51\% of functions and 60\% of global variables.

#### Partitioned architecture.
The Digital Bitbox functionality placed inside the enclave includes: (i) command processing functions for specific API calls; (ii) code for generating seeds (using the SGX-provided hardware random
Second, EBR is effective at reducing the number of \texttt{ecall} and \texttt{ocall} crossings at runtime, despite only moving a few additional functions into the enclave. In the case of Digital Bitbox, moving four functions into the enclave reduces the number of enclave boundary crossings by up to three orders of magnitude.

5.2 Performance evaluation

We evaluate the performance of the three partitioned applications in terms of throughput and latency.

**Experimental set-up.** All experiments are executed on an SGX-supported 4-core Intel Xeon E3-1280 v5 at 3.70 GHz with 64 GB of RAM, running Ubuntu 14.04 LTS with Linux kernel 3.19 and the Intel SGX SDK 1.7. We deactivate hyper-threading and compile the applications using GCC 4.8.4 with -O2 optimisations.

**Application benchmarks.** We evaluate Memcached with the YCSB benchmark [15]. Clients run on separate machines connected via a Gigabit network link. We increase the number of clients until the server is saturated. Memcached is initialised with the YCSB default of 1000 keys with 1 KB values. We then vary the percentage of get (read) and set (write) operations.

For LibreSSL, we measure the throughput and latency when signing certificates using SHA-256 and a 4096-bit RSA key. For Digital Bitbox, we observe the performance for the \texttt{seed}, \texttt{sign}, and \texttt{random} API calls using workloads from the Digital Bitbox test suite: (i) \texttt{tests\_sign} seeds a wallet and signs 64-byte transactions; (ii) \texttt{tests\_aes\_cbc} seeds a wallet with user-provided entropy, sets passwords and performs encryption/decryption with AES-256; and (iii) \texttt{tests\_random} returns random numbers.

**Results.** We measure the throughput and latency for Memcached: (i) partitioned by Glamdring; (ii) executed by SCONE (without network encryption); (iii) by Graphene; and (iv) natively, as the request rate is increased. We consider three workloads: read-only, write-only and 50%/50% read/write.

Fig. 3 shows that all three variants exhibit consistent behaviour across the workloads. Glamdring shows a throughput of 160k requests/s; SCONE (without encryption) achieves between 270k–330k requests/s; Graphene between 65k–95k requests/s; and the native Memcached achieves around 530k–600k requests/sec.

The reason for Glamdring’s lower throughput com-
Figure 4: Throughput and latency overhead for Digital Bitbox and LibreSSL

pared to SCONE is that SCONE avoids all enclave transitions; it trades off TCB size for performance (see §2.3) and requires user-level enclave threading to avoid transitions. Enclave transitions dominate the cost of processing a request; by batching multiple get requests together using multi-get, Glamdring achieves over 210k requests/sec. However, Glamdring has only a third of the TCB of SCONE (see Tab. 2). The overhead of a library OS means that Memcached with Graphene exhibits worse performance than Glamdring.

Fig. 4 shows the performance of LibreSSL and Digital Bitbox with Glamdring compared to their native versions. The throughput of certificate signing in LibreSSL is 0.6x compared to native execution, decreasing from 63 to 36 signatures per second on each CPU core. The performance of LibreSSL is limited by a single ecall (bn_sub_part_words), which is central to the RSA algorithm and accounts for 95% of all enclave transitions. As for Digital Bitbox, compared to native execution, the relative throughput is between 0.7x and 0.8x; the relative latency is between 1.3x and 1.4x.

Effect of EBR. By comparing the performance of partitioned applications before and after applying the EBR optimisation, we found that the latter increased the throughput by 1.6x to 4.0x for the three applications, at the cost of at most 4 additional functions in the enclave.

6 Related Work

Privilege separation. The attack surface of applications can be reduced in many ways [11, 31, 35, 51, 66, 85]. PrivTrans [11] performs a least-privilege partitioning of an application into a privileged monitor and an unprivileged slave component using static analysis, without considering the integrity of sensitive data. ProgramCutter [85] and Wedge [8] rely on dynamic analysis to partition applications. SeCage [51] combines static and dynamic analysis to partition applications, and the isolation is enforced using CPU virtualisation features. In contrast, Glamdring does not need a trusted OS or hypervisor and respects the constraints of trusted execution.

SOAAP [35] helps developers to reason about the potential compartmentalisation of applications based on source annotations and static analysis. Unlike Glamdring, it does not support automated code partitioning. Rubinov et al. [66] propose a partitioning framework for Android applications. It refactors the source code and adds a set of privileged instructions. However, it only supports type-safe Java applications and requires users to re-implement the security-sensitive functionality in C.

Protecting applications from an untrusted OS. A number of approaches have been proposed to deal with an untrusted OS that spans millions of LOC. NGSCB [57] and Proxos [79] execute both an untrusted and a trusted OS using virtualisation, and security-sensitive applications are managed only by the trusted OS. The TCB, however, still includes a full OS. In more recent work, Overshadow [13], SP³ [87], InkTag [39] and Virtual Ghost [20] protect application memory from an untrusted OS by extending the virtual machine monitor (VMM). Such approaches put trust in the VMM, and cannot protect against attackers with privileged access, such as system administrators.

Trusted hardware. Use of trusted hardware, such as secure co-processors [50] and trusted platform modules (TPM) [80], can protect against attackers with physical access. A TPM can measure system integrity and provide remote attestation to verify the software stack [29]. Since the TPM measurement will include the OS and any system libraries, the TCB likely comprises millions of LOC.

Flicker [53] reduces the integrity measurement to a TCB of just 250 LOC, but lacks relevant system support and suffers from slow TPM operations. TrustVisor [52] is a special-purpose VMM that uses software-based μTPMs for application integrity checking, but it focuses on small pieces of application logic and requires a trusted hypervisor. CloudVisor [88] provides integrity and confidentiality protection for virtual machines using nested virtualisation, but this leads to VM-sized TCBS.

7 Conclusions

We described Glamdring, the first partitioning framework that helps developers leverage SGX enclaves for C applications. Glamdring uses static program analysis to decide which subset of the application code to protect, and offers guarantees that the confidentiality and integrity of application data cannot be compromised, even when an attacker has complete control over the machine. Our experimental evaluation demonstrates that Glamdring is sufficiently practical to handle real-world applications.

8 Acknowledgements

This work has received funding from the European Union’s Horizon 2020 programme under grant agreements 645011 (SERECA) and 690111 (SecureCloud), and from the UK Engineering and Physical Sciences Research Council (EPSRC) under the CloudSafetyNet project (EP/K008129) and the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) (EP/L016796/1).
References


Ramalingam, G. The Undecidability of Aliasing. TOPLAS (1994).


