STELLA: Sparse Taint Analysis for Enclave Leakage Detection

Yang Chen‡, Jianfeng Jiang‡, Shoumeng Yan‡, Hui Xu‡
†School of Computer Science, Fudan University
‡Ant Group
China

ABSTRACT
Intel SGX (Software Guard Extension) is a promising TEE (trusted execution environment) technique that can protect programs running in user space from being maliciously accessed by the host operating system. Although it provides hardware access control and memory encryption, the actual effectiveness also depends on the quality of the software. In particular, improper implementation of a code snippet running inside the enclave may still leak private data due to the invalid use of pointers. This paper serves as a first attempt to study the privacy leakage issues of enclave code and proposes a novel static sparse taint analysis approach to detect them. We first summarize five common patterns of leakage code. Based on these patterns, our approach performs forward analysis to recognize all taint sinks and then employs a backward approach to detect leakages. Finally, we have conducted experiments with several open-source enclave programs and found 78 vulnerabilities previously unknown in 13 projects.

CCS CONCEPTS
• Computer systems organization → Embedded systems; Redundancy; Robotics; • Networks → Network reliability.

KEYWORDS
taint analysis, enclave, sparse analysis, privacy leak

ACM Reference Format:
Yang Chen†, Jianfeng Jiang‡, Shoumeng Yan‡, Hui Xu†. 2022. STELLA: Sparse Taint Analysis for Enclave Leakage Detection. In Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX). ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION
Due to the advantages of cloud computing, migrating on-premise software to public cloud services has received more and more attention[10]. However, privacy is still a major concern for many potential cloud users that cannot afford data leakage risks[9]. TEE (trusted execution environment) is a promising solution to this problem provided by chip manufacturers, such as Intel Software Guard Extensions (SGX) [13, 26] and ARM TrustZone [14, 28]. With TEE, cloud users can deploy their security-critical applications in isolated enclaves and prevent them from being accessed by unauthorized parties. In recent years, we have witnessed the application of Intel SGX in many complex systems, including database [30, 40, 47], middleware [6, 21], blockchain [5, 27], and network [4, 29].

Although Intel SGX provides hardware-level protection for the software, its practical effectiveness largely depends on how users code their programs. Any improper use of the SGX interface or bugs in the enclave code may lead to privacy leakage [31]. To our best knowledge, there is still neither systematic work on this problem nor available solutions to detect such bugs. Studying this issue is particularly important because verifying the effectiveness of TEE code is a critical step to the advancement of TEE solutions.

In this study, we first list five bug patterns that lead to privacy leaking in enclave programs (writing sensitive data to ECALL out, ECALL user_check, OCALL in, OCALL return and NULL pointers). To the best of our knowledge, no research has summarized these patterns. A well-known technique for finding privacy leaks is taint analysis [2, 7, 42], and it is straightforward to apply taint analysis to find the bug patterns we define. However, enclave programs’ privacy leak exhibits a variety of traits. For instance, taint sinks are data writes to particular pointers that are unknown and require solving. This is distinct from the typical one in that taint sinks are fixed and well-known APIs. Therefore, it is challenging to use taint analysis to tackle this issue.

To find these patterns in enclave programs, we provide an innovative custom taint analysis. We initially investigate the def-use relationship of the variables in enclave programs in order to construct a value flow graph (VFG). In order to identify the data writes to these pointers (i.e., identify taint sinks), we first perform forward analysis starting from the definition nodes of these pointers on the VFG. The written data is examined using backward analysis to determine its sensitivity. We employ VFG rather than control flow graph (CFG) as in conventional static analysis because VFG-based analysis (also known sparse analysis) has been proven in prior studies [33, 34, 36–38] to improve performance without sacrificing accuracy.

To elaborate, the Enclave Definition Language (EDL) [15] files, which serve as interface descriptions, and the LLVM IR files generated by the enclave program are the inputs used by our approach. We parse EDL files to extract pointer parameters that could result in the leakage of private information and to create the VFG, we read the LLVM IR files. Then, starting with the definition nodes associated with the aforementioned pointer parameters, we traverse
the VFG and taint the variables in accordance with the rules we established for taint propagation. We designate them as taint sinks if we discover writes to these taint variables in the enclave program. The VFG also provides the corresponding nodes for the data written in a taint sink. To obtain all the leaked variables, we begin from these nodes and move backward via the VFG. If any of them are private, it suggests a possible privacy breach. Our technique provides manual insensitive annotation at the source code level and insensitive variables will be automatically omitted in the backward analysis to reduce false positives because not all variables in the enclave program are sensitive. Furthermore, the decrypted data, the data to be encrypted, and the key for encryption or decryption will be automatically tagged as high-risk sensitive data. In the leak report, the high-risk data will be shown with a greater priority if it has been compromised.

We have implemented a prototype for the Sparse Taint analysis for Enclave privacy Leakage detection, namely STELLA. In addition to employing sparse analysis to increase speed, we have adopted a preference for retrieving the call graph (CG) rather than the call flow graph (CFG) for solving variables on VFG that could reveal secrets because CFG has a significantly higher number of nodes than CG. We used STELLA to analyze 13 open-source SGX-based applications on GitHub to test the viability of our approach, and we discovered 78 privacy disclosure bugs there that had never before been reported.

In short, this article contains several major research contributions as follows.

- Privacy leaks caused by coding errors in enclaves undermine enclave confidentiality guarantees, making it easier for attackers to obtain secrets in enclaves. To the best of our knowledge, our work is the first comprehensive one on this issue.
- We first define patterns of privacy-leaking coding mistakes that enclave developers are prone to make and then propose a novel sparse taint approach to efficiently analyze the vulnerabilities in enclave code based on these patterns. As far as we know, this is the first available approach on the issue.
- We have implemented a prototype tool, STELLA, and released it as open-source on GitHub. Our experimental results show that STELLA can effectively discover privacy-leaking vulnerabilities in enclave programs.

The rest of our paper is organized as follows. We introduce the Intel SGX background in section 2. Section 3 defines the problem of privacy leakage on enclave programs and discusses the challenges of detecting such bugs. Section 4 then elaborates on our sparse taint analysis approach. Section 5 evaluates the performance of our approach. We review related work in Section 6. Section 7 concludes our paper.

2 PRELIMINARY

In this section, we first introduce background about SGX and Intel SGX SDK, then illustrate the privacy leakage issue when developing SGX programs.

![Figure 1: The underlying mechanism of SGX.](image)

2.1 SGX Background

Intel SGX is a set of CPU instructions that enable applications to create hardware-based protected areas, namely enclaves. Enclaves are used to protect private data from modification and closure [43]. Private data within an enclave can only be accessed by the code within the same enclave, and cannot be read or written by programs outside the enclave even such programs run at privilege levels.

A big change in introducing enclaves in a normal program is that we need to separate the application into the untrusted part (outside enclave) and trusted part (inside enclave) and all access to secrets needs to be done in the enclave. As shown in Figure 1, untrusted code can create one or more enclaves. When the secrets need to be processed, the untrusted code needs to call the trusted function like a normal function, and then the CPU is switched to the enclave mode via the EENTER instruction, and the control is transferred to the corresponding trust function. The trusted function in the enclave can directly access these secrets. After processing these secrets, the CPU switches out of enclave mode via EEXIT instruction, and the enclave transfers control to untrusted code and continues normal execution.

2.2 Intel SGX SDK

Intel SGX SDK [16] is a set of toolkits to develop enclave programs. In practice, developers tend to write programs with high-level languages, instead of directly using low-level instructions like EENTER and EEXIT. To this end, SGX SDK provides high-level abstractions on low-level instructions and enables developers to develop enclave programs with C/C++. Currently, enclave programs are mostly developed with Intel SGX SDK.

Intel SGX SDK introduces ECALLs (enclave calls, the calls to enclave functions) and OCALLs (outside calls, the calls to outside-enclave functions) on top of the EENTER and EEXIT instructions. An ECALL calls the EENTER instruction first, then executes the trusted function, and finally calls the EEXIT instruction. OCALL is vice versa. ECALLs and OCALLs are the interfaces between host
and enclave, and their prototypes are defined in EDL files. As shown in Figure 2, we declare an ECALL foo and an OCALL bar in the EDL. The trusted ECALL foo is defined in the enclave. The untrusted OCALL is defined in the host.

Intel SGX SDK automatically generates proxy and bridge functions for ECALLs and OCALLs when compiling enclave programs. Take the ECALL foo in Figure 2 as an example, we call its definition in enclave real foo(). Intel SGX SDK will generate a pair of functions, an untrusted proxy and a trusted bridge function. When executing this ECALL as Figure 3 shows, the host code first calls the untrusted proxy. Then the untrusted proxy will call the trusted bridge. Finally, the trusted bridge calls the real foo(). The state and return value of the ECALL are propagated back from the opposite direction.

There are two categories of pointer parameters for functions defined in EDL files, namely directed pointers, and raw pointers.

Pointers in ECALLs and OCALLs can be declared with direction (in, out) and size (size_t). The proxy and bridge functions will copy the content of pointers based on the direction and size. Take foo in Figure 2 as an example, in_ptr has the in attribute, and its size attribute is 10. The trusted bridge function will allocate a 10-bytes buffer inside the enclave and copies the first 10 bytes of in_ptr to the buffer. Similarly, if the direction is out, the trusted proxy will copy data from the enclave to the the untrusted world.

Pointers can also be annotated as user_check (e.g. uc_ptr in foo in Figure 2), which means it is a raw pointer. The bridge or proxy will not copy the buffer but directly pass the address. Note that the pointer returned by an OCALL, like bar in Figure 2, is also a raw pointer, and developers need to do proper checks before using it. In addition, if an in pointer is a pointer to struct, since the buffer copy is a shallow copy, the pointer fields in the struct are all user_check pointers.

In general, the pointer direction attributes in EDL determine whether buffer copying or raw pointer transfer occurs between host and enclave. If developers are not careful with these pointers, sensitive data can be accidentally leaked into the untrusted world. In the following subsection, we will demonstrate this problem with a real-world example.

2.3 Issues of Developing with SGX SDK

Although SGX prevents host code from directly accessing data inside the enclave, it is still possible to leak data outside the enclave through the pointer parameter of ECALL or OCALL. For example, writing sensitive data to the in pointers in OCALLs may cause privacy leakage.

Such privacy leakage problems are detected in existing enclave programs. BiORAM-SGX [17] is a personal genetic data statistical
analysis system using SGX. The system claims that sensitive personal genetic data will not be leaked to the server. But we found the problem of AES key leakage in this system. Figure 4 demonstrates the buggy code snippet. data of OCALL_SaveFile is an OCALL in pointer (line 2), which can pass information from the enclave to the outside of the enclave. The developers must be careful with this pointer and cannot pass confidential information to data. However, ECALL_VCFEncryption passes the AES key to data in plaintext without sealing, calling OCALL_SaveFile to save the AES key out of the enclave (line 12). An attacker can easily open the unencrypted file and obtain the AES key, which poses a great threat to user data security.

Note that this vulnerability does not contradict SGX’s guarantee against data leakage. SGX SDK requires developers to perform checks when writing sensitive data to pointers. If the developers carelessly fail to check, SGX cannot guarantee that the data inside the enclave will not be leaked.

3 PROBLEM AND CHALLENGES
For enclave programs, developers may mistakenly copy sensitive data out of the enclave, which will cause privacy leakage. Note that the privacy leakage problem is serious for enclave programs because this problem breaks SGX’s guarantee of data security. SGX aims to protect private data from access from the host and not allow the leaking of private data to the untrusted world.

Next, we first summarize common code patterns that may cause the problem, then illustrate the challenges to adapt taint analysis to enclave programs.

3.1 Typical Patterns
For enclave programs, most privacy leakages are caused by writing sensitive data to specific pointers. As shown in Figure 5, by comprehensively studying the coding specification of Intel SGX SDK, we summarize 5 pointer usage patterns that can leak privacy in enclave codes. To the best of our knowledge, no one has yet summarized these patterns.

Write Private Data to ECALL out Pointers (P1). As shown in Figure 3, When ECALL foo returns, the data written to the ECALL out pointer is copied into untrusted memory by the bridge function. Therefore, when a developer mistakenly writes private data to an ECALL out pointer, the private data is leaked.

Write Private Data to ECALL user_check Pointers (P2). To achieve better performance, developers may use user_check pointers in ECALL to avoid checking pointers and copying buffer by SGX SDK. Instead, developers need to check the user_check pointer by themselves. But they may carelessly not check user_check pointers and write sensitive data to them. If the pointers point to untrusted memory, private data may leak. Note that we cannot trust the ECALL input according to the Intel SGX threat model, so even if enclave developers assume that these pointers point inside the enclave, they may also be tampered with by an attacker to point to untrusted memory. Note that the pointer field in the in structure pointer is user_check, so this pattern also includes this case.

Write Private Data to OCALL in Pointers (P3). Figure 3 shows how Intel SGX SDK processes the in pointer in ECALL. The bridge function copies the untrusted memory data to the enclave. For the in pointer of OCALL, the processing is similar but in the opposite direction. The Intel SGX SDK will copy the enclave memory data to the untrusted memory, so developers must be careful, if the OCALL in pointer points to sensitive data, it will lead to privacy leakage.

Write Private Data to OCALL Return Pointers (P4). If the return pointer of OCALL points to untrusted memory, and the developer mistakenly writes sensitive data to this pointer, it will lead to privacy leakage.

Write Private Data to NULL Pointers (P5). For general applications, writing data to address 0 (NULL) will be aborted by the OS. However, for SGX applications, the OS is untrusted according to the Intel SGX threat model. Writing sensitive data to NULL pointers in the enclave can lead to privacy leakage.

Taint analysis is a traditional method to detect such patterns. However, since enclave programs differ from traditional programs in many ways, we have to adapt the taint analysis to enclave programs.

3.2 Challenges
The traditional method to detect privacy leakage is via taint analysis, e.g., for android or IoT programs. However, traditional taint analysis cannot be directly ported to detecting privacy leakage on enclave programs. A traditional taint analysis framework includes taint sources, propagation rules, and taint sinks. Enclave programs differ from android or IoT programs in the following three aspects:

Taint sources are variables storing sensitive data. In general, android programs only have a few taint sources such as those variables storing passwords and phone numbers so manually annotating such variables is easy. For enclave programs, considering that SGX protects all data inside the enclave, we should treat all of them except a little public data as taint sources. Hence, there are two main challenges to our task. First, the taint sources will be
4 SPARSE TAINT ANALYSIS APPROACH

In general, we use static analysis on SGX programs to detect privacy leakage vulnerabilities. The whole STELLA framework is shown in Figure 6. STELLA receives EDL files and annotated enclave code as input and generates a leakage report. Preprocessing and analysis module comprise the STELLA pipeline.

4.1 Preprocess

As shown in Figure 6, STELLA first preprocesses input files, including constructing graphs, extracting key function parameters, and identifying insensitive data. The module accepts compiled bitcode and EDL files as input and outputs graphs and tables required for further analysis.

4.1.1 Construct Graphs. We construct several graphs for analysis, including CG and VFG. STELLA bases graph construction on SVF [37], a static value flow analysis framework. VFG plays a key role in our analysis. STELLA performs an inter-procedural pointer analysis to obtain the points-to information after reading the enclave bitcode files. With the points-to information, STELLA constructs a memory SSA form and obtains the def-use chains and value flows to construct the VFG.

Figure 7 demonstrates CG and VFG of an example program. Figure 7a is a simple taint source-sink example program. The program fetches the user name and password, assembles a message, and finally sends it to the outside via HTTP, causing sensitive information leakage. Call graph (CG) describes the calling relationship between functions. As shown in Figure 7b, the main function calls getUsername, getPassword, format and sendHTTP. Value Flow Graph (VFG) demonstrates the data flow of variables. As shown in Figure 7c, the values of variables usrm and passwd both flow to variable msg. Our approach is based on CG and VFG. In general, we begin by matching CG nodes with the function name. The taint sink nodes on the VFG are then discovered using these nodes. Finally, leakage of personal information is discovered by resolving the graph reachability problem between sinks and sources.

4.1.2 Extract Key Parameters. We extract the function parameters in ECALLs and OCALLs in EDL files that may potentially leak privacy. For each parameter, we combine its function name, the position index, and the leak pattern into a tuple. For example, for an ECALL, ecall_func[(user_check) void* uc_ptr, [out, size = size] void* out_ptr, int int size], uc_ptr and out_ptr are two parameters that may leak privacy, so we will get two tuples, (ecall_func, 0, P2) and (ecall_func, 1, P1). After scanning all EDL files, we will get a list of tuples, namely tuplelist. It is used for subsequent taint analysis to find sinks.

4.1.3 Identify insensitive Data. We manually annotate some insensitive variables in enclave programs. Since enclaves are used to protect private data, we assume that most data processed inside enclaves is sensitive. However, some variables storing insensitive information are used to interact with the untrusted world, like user_id in Figure 8. We will annotate this variable with the
Annotate the insensitive variable user_id with INSENSITIVE. Other variables are sensitive by default.

INSENSITIVE prefix. We assume all variables without the annotation store sensitive data. 

The instruction to write data to the pointer in the search result is the taint sink.

Pointers from Outside. From an enclave developer’s perspective, ECALL out (P1), ECALL user_check (P2), and OCALL return pointers (P4) all come directly from outside the enclave. The strategies for detecting their associated taint sinks are also similar. Note that the enclave code may derive new pointers from these pointers. For example, if an ECALL passes in a user_check secondary pointer (e.g. [user_check]void** p1), then this pointer is dereferenced to get a new pointer void* p2 = *p1. p2 may also point outside the enclave. Writing sensitive data to p2 may also lead to privacy leakage. Therefore, to make our analysis as sound as possible, we search not only these pointers but also the pointers derived from them via a taint-style algorithm.

The instruction to write data to the pointer in the search result is the taint sink.

Pointers Declared Inside. Pointers defined inside the enclave can also be corrupted, such as null pointers or wild pointers. These pointers may point outside the enclave. We define writing instructions to these pointers as taint sinks. We focus on the privacy disclosure vulnerability caused by writing sensitive data to a NULL pointer in an enclave (i.e., P5).

Explicit Sinks. Calling OCALL functions that accept primitive types or in pointers as parameters can leak private data (i.e., P3). We call this type of taint sinks Explicit Sinks because they can be located directly by matching function names in the call graph.

Algorithm 1: Taint sinks finding

| Function FindSinks(EDLs, prog) |
|--------------------------------|
| tupleList ← ExtractKeyParameters(EDLs) |
| cg, cfg ← ConstructGraphs(prog) |
| sinks ← {} |
| for tuple ∈ tupleList do |
|   /* match the cg node by funcname. */ |
|   node ← GetNode(tuple.funcname, cg) |
|   /* Pointers from Outside (ECALL out / user_check / OCALL return)*/ |
|   if tuple.leafType ∈ (P1, P2, P4) then |
|     ptr ← GetPtr(node, tuple.index) |
|     /* ptrTaint is shown in Algorithm 2 */ |
|     sinks ← sinks ∪ PtrTaint(ptr) |
|   /* Explicit Sinks (OCALL in pointers)*/ |
|   else if tuple.leafType ∈ (P3) then |
|     ptr ← GetPtr(node, tuple.index) |
|     defNode ← GetDefNode(ptr, cfg) |
|     sinks ← sinks ∪ (defNode) |
|   /* for NULL pointers (P5) */ |
|     /* malloc without check */ |
|     /* node ← GetNode("malloc", cg) */ |
|     ptr ← GetPtr(node) |
|     compile_instrs ← GetCompileInstrs(ptr) |
|     if (compile_instrs == 0) then |
|       sinks ← sinks ∪ PtrTaint(ptr) |
| return sinks |
We have implemented a prototype tool STELLA. STELLA can detect whether the nodes in the paths are variable allocation instructions (e.g. alloc). If so, and they are not yet in the InsensitiveTable, then we add the source code location of the sink and leaked variable to the report.

5 EVALUATION

We have implemented a prototype tool STELLA. STELLA can detect the patterns that cause privacy leakage for enclave programs. We perform evaluation experiments to evaluate the effectiveness and

\[
T(x), z = op(x, y) \\
T(z)
\]

(4)

\[
T(\text{base}), \text{addr} = \text{gep(base)} \\
T(\text{addr})
\]

(5)

\[
T(x), y = \text{bitcast(x)} \\
T(y)
\]

(6)

In the formula (1), the LLVM load instruction refers to reading a value from memory. When \( \text{addr} \) is a tainted variable, then the value \( v \) read from \( \text{addr} \) is also marked as a tainted variable.

In the formula (2), the store instruction is used to write memory. The store instruction has two arguments: a value \( \text{addr} \) and \( v \), the patterns that cause privacy leakage for enclave programs. We then add the source code location of the sink and leaked variable in the reverse direction along the def-use chain, and get all the sensitive. Algorithm 3 shows our backward tracking algorithm. We backward track leaked variables and determine whether they are sensitive. Algorithm 3 shows our backward tracking algorithm. We process each taint sink, traverse from the VFG node of the sink in the reverse direction along the def-use chain, and get all the paths from the sink to the leaf nodes. Then, we start to find out whether the nodes in the paths are variable allocation instructions (e.g. alloc). If so, and they are not yet in the InsensitiveTable, then we add the source code location of the sink and leaked variable to the report.

4.2.3 Backward tracking and sensitive information determination. After taint sink finding, we can obtain all code snippets in the enclave that transmit information to the untrusted world. Next, we backward track leaked variables and determine whether they are sensitive. Algorithm 3 shows our backward tracking algorithm. We process each taint sink, traverse from the VFG node of the sink in the reverse direction along the def-use chain, and get all the paths from the sink to the leaf nodes. Then, we start to find out whether the nodes in the paths are variable allocation instructions (e.g. alloc). If so, and they are not yet in the InsensitiveTable, then we add the source code location of the sink and leaked variable to the report.


| Project Name                          | Description                                | ECALL user_check | ECALL out | OCALL in | OCALL return | Null ptr | Total |
|--------------------------------------|--------------------------------------------|------------------|-----------|----------|--------------|----------|-------|
| SGX-Tor [19]                        | secure anonymity network                    | 1                | 1         | 7        | 9            | 1        | 2     |
| sgx_wechat_app [46]                  | trusted wechat app                         | 1                | 1         |          | 1            | 1        | 3     |
| Fidelius [12]                       | protect browser users’ secrets              | 1                | 1         |          | 5            | 6        | 11    |
| sgx-based-mix-networks [11]         | hidden anonymization                       | 1                | 1         |          | 1            |          | 3     |
| sgx-dnet [44]                       | machine learning inside enclave            | 1                | 1         |          | 25           | 26       | 51    |
| SGX_SQLite [25]                     | secure SQLite database                      | 1                | 1         |          |              |          | 2     |
| TaLoS [3, 29]                       | secure TLS library                         | 2                | 1         |          |              | 3        | 6     |
| sgx-aes-gcm [32]                    | SGX AES-GCM usage example                   | 1                | 1         |          |              |          | 2     |
| password-manager [22]               | password manager using SGX                  | 1                | 1         |          | 2            |          | 4     |
| TACIoT [39]                         | IoT data protection                        | 1                | 1         |          | 2            |          | 4     |
| BiORAM-SGX [17]                     | genome analysis system                     | 2                | 2         |          |              |          | 4     |
| PrivacyGuard [41]                   | data analytic inside enclave               | 2                | 7         |          | 9            |          | 18    |
| Town-Crier [45]                     | smart contract using SGX                   | 2                | 12        |          | 14           |          | 30    |
| **Total**                           |                                            | 2                | 2         | 14       | 2            | 58       | 78    |

5.1 Experimental Setup

We perform experiments to study two research questions:

**RQ1:** Can STELLA effectively find privacy leakage bugs in enclave programs?

**RQ2:** How efficient is STELLA when analyzing real-world projects?

We use STELLA to test the enclave programs developed using the SGX SDK on GitHub. We found bugs in 13 popular projects (the average stars of these projects is 30). Table 1 shows the details of these projects. The code size of these projects varies largely, ranging from a few hundred lines to 527K lines.

Our experiments are done on a ubuntu-20.04 server with an Intel i7-9700T 4.30GHz 8-core CPU. The version of Intel SGX SDK is 2.15. The reported run time is the average of three measurements.

5.2 Effectiveness

5.2.1 Overview. STELLA found 78 vulnerabilities in these projects that could leak sensitive data. Table 1 shows the number of vulnerabilities with different patterns. For example, we found that SGX-Tor has two privacy leakage bugs, one of which is to write sensitive data to the OCALL in pointer, and the other is to write sensitive data to the pointer returned by OCALL.

5.2.2 Patterns. We find all five patterns in real-world enclave projects. Among all patterns, writing private data to OCALL in pointers and to NULL pointers are the most common patterns. Next, we will present a case in real-world projects for each pattern.

Write Private Data to ECALL out pointers (P1). Figure 9 shows a secret disclosure bug in sgx-based-mix-networks. The enclave code incorrectly writes the secret to the ECALL out pointer. From the EDL file, we can know that the parameter result is an out pointer. In dispatch function, message is sensitive data. But at line 12, message is directly copied to result without encryption. When the dispatch function returns, the message plain text in result will be copied to the untrusted content (line 2), causing sensitive information leaked.

```c
1 //***application code***/
2 unsigned char content[KEY_SIZE];
3 ...
4 dispatch(this->ed, &dispatch_response, content, buffer_size, this->fan_out, 
5       &buffer_size, this->fan_all_out);
6 //***enclave code***/
7 int dispatch(unsigned char *result, int *fan_out, size_t 
8       *buffer_size, int fan_all_out)
9 {
10   ...
11   if (*fan_out) {
12     // unencrypted message leaked !!!
13     std::copy(message, message + strlen((char *)message), 
14       result);
15     return 1;
16   }
17   ...
18   sgx_status_t ret_encrypt = 
19       sgx_rsa_pub_encrypt_sha256(previous_public_key, result,  
20       out_len, message, strlen((char *)message));
21   ...
22 }
```

Figure 9: A P1 privacy leakage vulnerability in sgx-based-mix-networks. The enclave code mistakenly writes unencrypted sensitive information to the ECALL out pointer (line 12)

Write Private Data to ECALL user_check pointers (P2). TaLoS [29] is a library that enables applications to terminate TLS connection securely. TaLoS protects sensitive data from disclosure by placing sensitive data within a SGX enclave. But for this library, there is still a hidden danger of SSL private key leakage. Figure 10 demonstrates the code snippet that may cause data leakage. Function eccall_SSL_get_privatekey is an ECALL and the parameter

1https://anonymous.4open.science/r/STELLA-7724
Figure 10: A P2 privacy disclosure vulnerability in TaLoS that leaks an SSL private key. pkey is an ECALL user_check pointer, the enclave code erroneously writes the private key enclave_pkey to pkey. TaLoS developers have confirmed the vulnerability.

```c
void ecall_SSL_get_privatekey(EVP_PKEY* pkey, SSL *s) {
  EVP_PKEY* enclave_pkey = SSL_get_privatekey(in_s);
  // Leak !!!
  memcpy(pkey, enclave_pkey, sizeof(*pkey));
  ...
}
```

Figure 11: The P3 privacy disclosure vulnerability in sgx_wechat_app will reveal the sgx_ra_key and the decrypted mobile phone number. The enclave code mistakenly passes sensitive information to the OCALL in pointer.

```c
/****IDL****/
uve_trusted {
  ...
  void ocall_eprintf_string(uv_in, str);
  ...
};
/****ocall codes****/
sgx_status_t ecall_decrypt_secret(sgx_ra_ctxt_t context,
  const uint8_t *p_secret, uint32_t src_len,
  const sgx_aes_gcm_128bit_tag_t *p_in_mac) {
  sgx_ra_key128_t ra_key;
  sgx_status = sgx_ra_get_keys(context, SGX_RA_KEY_SK, &ra_key);
  // ra_key is leaked !!!
  fprintf("ra_key:0x\n", hexstring(ra_key, sizeof(ra_key)));
  ...
  sgx_status = sgx_rjndael256GCM_decrypt(ara_key, p_src, src_len, p_dst, p_iv, SGX_AESGCM_IV_SIZE, NULL, a, p_in_mac);
  phone_num = string(hexstring(p_dst, src_len));
  accid_phone_map[context][accid_map[context]] = phone_num;
  // the decrypted phone_num is leaked !!!
  fprintf("Phone number:0x\n", accid_phone_map[context][accid_map[context]]);
  ...
} int eprintf(const char *fmt, ...) {
  ...
  fprintf(buf, BFSIZE, fmt, ap); // fmt flows into buf
  ocall_eprintf_string(buf); // buf is passed to OCALL in ptr
  ...
} /* app codes */
void ocall_eprintf_string(const char *str){
  printf_info(felog, "%s", str); // print in untrusted world.
}
```

Write Private Data to OCALL in Pointers (P3). As shown in Figure 11, the function ecall_decrypt_secret in the enclave converts the variable ra_key to hexadecimal format and passes it to the function fprintf at line 16. This ra_key is sensitive data. At line 22, the just decrypted phone_num is also passed to fprintf. In fprintf function, the ra_key and phone_num will be copied into the buf at line 27. Then at line 28, OCALL ocall_eprintf_string is called by passing buf to its in pointer parameter. During the execution of ocall_eprintf_string, Intel SGX SDK will copy the sensitive data in buf outside enclave (line 34). ra_key and phone_num are finally printed directly on the standard output of the untrusted OS, so attackers can easily get them with little effort.

Write Private Data to OCALL Return Pointers (P4). Figure 12 shows a privacy leakage vulnerability in TaLoS. ocall_malloc is an OCALL that returns a pointer (line 3). In the enclave code, ssl_update_cache updates the cache and calls ocall_new_session_callback_wrapper (line 10). Then, ocall_new_session_callback_wrapper will call the OCALL ocall_malloc when ssl_session_outside is NULL to apply for a block of untrusted memory (line 17), and write the SSL session to this memory (line 20), resulting in the disclosure of SSL security information.

Write Private Data to NULL Pointers (P5). As shown in Figure 13, PrivacyGuard's enclave code first uses malloc to allocate a block of trusted memory (line 4) but forgets to check whether the result is successful, so DO_data_key may be NULL, and then generates a 16-byte random number as the key. The key is stored in DO_data_key (line 11), if the memory allocation fails, the key will be leaked. The developers of PrivacyGuard have confirmed the vulnerability.

```c
sgx_status_t ECALL_enclave_DO_config(int num_ENDs){
    ...
    // malloc DO_data_key but not check the result.
    DO_data_key = (sgx_aes_gcm_128bit_key_t *) malloc(num_ENDs * sizeof(sgx_aes_gcm_128bit_key_t));
    ...
} /* app codes */
void ocall_eprintf_string(const char *str){
  printf_info(felog, "%s", str); // print in untrusted world.
}
```

Figure 13: A P5 privacy leakage vulnerability in PrivacyGuard. The enclave code uses malloc to allocate a block of memory to store the encryption key, but forgets to check if the return pointer is NULL. When the host runs out of memory or is attacked, the encryption key can be compromised.
5.3 Performance

To answer RQ2, we measured the time it took STELLA to analyze these open-source enclaves, and the results are shown in Table 2. Analysis time is positively related to the number of lines of codes. When the number of lines of code is about 10K, the analysis speed of STELLA is fast, and the analysis can be completed within 10 seconds. When the number of lines of code is around 500K (TaLoS), the analysis time goes up, but it does not exceed 10 minutes. Overall, STELLA is very efficient even when analyzing large programs.

5.4 Threats to Validity

False positives are not statistically analyzed in the experiment. Many leaks in the report are difficult to distinguish whether it is sensitive information because this requires more prior knowledge of the enclave programs. The bugs in Table 1 have been manually confirmed to be high-risk sensitive information. In addition, our NULL pointer detection is not sound. We perform pointer analysis [1] (the point-to set of NULL pointer is empty) and malloc analysis (whether the enclave code assumes that malloc will succeed), so the actual numbers of P5 vulnerabilities may be more and the problem is more serious. Furthermore, we cannot detect leaks caused by wild pointers and mathematically manipulated pointers outside the enclave.

6 RELATED WORK

Several previous research also studies privacy leakage problems in enclave programs. COIN attacks [18] summarizes four interface-oriented attacks: Concurrent, Order, Inputs, and Nested, and implements a testing framework to detect bugs with instruction emulation and concolic execution. COIN detects enclave memory information leakage by checking the length of memcpy or a loop condition. This approach can only detect privacy leakage caused by the out-of-bounds copy, but cannot detect the patterns we defined.

TeeRex [8] mainly detects memory corruption vulnerabilities in the enclave code introduced by the interface between the host and the enclave. These vulnerabilities could allow attackers to corrupt function pointers and arbitrary memory writes. In terms of enclave information leakage, TeeRex’s work is relatively limited, and it only briefly explains that under the vulnerability of null pointer dereference, malicious user_check pointer input causes arbitrary memory read. However, our work demonstrates that even in the absence of malicious third parties, the enclave code may be leaking secrets.

Moat [35] employs formal verification to verify whether the enclave code leaks secrets to an adversary. However, Moat is not flexible and scalable enough to apply to large real-world enclave code. STELLA can efficiently analyze large open-source projects.

DEFLECTION [23] verifies the enclave programs by employing compiler instrumentation to insert some privacy security policies into the enclave programs. DEFLECTION introduces runtime overhead for enclave code. In the worst case, the performance overhead is 39.8%.

7 CONCLUSION

This paper investigates possible privacy leakage in enclave code. The main challenge lies in how to effectively and efficiently identify...
the privacy leakage code. We at first define five common privacy-leaking patterns, then propose a novel sparse taint analysis method to identify these leaking patterns. We implement a prototype STELLA and analyze several open-source enclave programs on GitHub with STELLA. Our experimental results show that sparse taint analysis can effectively and efficiently detect privacy leaking bugs. We believe that our method will shed light on further research on enclave privacy protection.

ACKNOWLEDGEMENT

This work was sponsored by CCF- AFSG Research Fund (project id: RF20210017).

REFERENCES

[1] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. Ph.D. Dissertation, Citeseer.
[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Barthe, Syed Zafar, Roman Eberle, and Vivekenthiran. 2017. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM Sigplan Notices 49, 6 (2014), 259–269.
[3] Pierre-Louis Aublin, Florian Kelbert, D O’Keffe, Divya Muthukumaran, Christian Fritz, Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Barthe, Syed Zafar, Roman Eberle, and Vivekenthiran. 2017. TaLoS: Efficient TLS Termination Inside SGX Enclaves for Existing Applications. https://github.com/lsds/TaLoS.
[4] Pierre-Louis Aublin, Florian Kelbert, D O’Keffe, Divya Muthukumaran, Christian Fritz, Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Barthe, Syed Zafar, Roman Eberle, and Vivekenthiran. 2017. TaLoS: Secure and transparent TLS termination inside SGX enclaves.
[5] Ghadebo Ayode, Vishal Karande, Latifur Khan, and Kevin Hamlen. 2018. Decentralized IoT data management using blockchain and trusted execution environment. In 2018 IEEE International Conference on Information Reuse and Integration (IRI). IEEE, 15–22.
[6] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrod
[43] Yuan Yu, Andy Zhao, and Lili Zhang. [n.d.]. Intel Software Guard Extensions SDK for Linux. Retrieved Apr 28, 2022 from https://01.org/intel-software-guard-extensions

[44] Peterson Yuhala, Pascal Felber, Valerio Schiavoni, and Alain Tchana. 2021. SGX-Darknet: SGX compatible ML library. https://github.com/SabaEskandarian/Fidelius.

[45] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. https://github.com/black5un/Town-Crier.

[46] Yao Zhao. 2020. Use sgx to resolve wechat app trust problem. https://github.com/TonyCode2012/sgx_wechat_app.

[47] Wenchao Zhou, Yidan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. Veridb: an SGX-based verifiable database. In Proceedings of the 2021 International Conference on Management of Data. 2182–2194.