2024 IEEE 11th International Conference on Cyber Security and Cloud Computing (CSCloud) | 979-8-3503-7698-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/CSCLOUD62866.2024.00029

COMURICE: Closing Source Code Leakage in Cloud-based Compiling via Enclave

1st Dahan Pan
Shanghai Jiao Tong University
Shanghai, China
dhpan98@sjtu.edu.cn

2nd Jianqiang Wang
CISPA
Saarbrucken, Germany
wjq.sec@gmail.com

3rd Yingpeng Chen
Shanghai Jiao Tong University
Shanghai, China
chen_yp@sjtu.edu.cn

4th Donghui Yu
Shanghai Jiao Tong University
Shanghai, China
yudhui@sjtu.edu.cn

5th Wenbo Yang
Shanghai Feysh Technology Co.,Ltd
Shanghai, China
wbyang@feysh.com

6th Yuanyuan Zhang*
Shanghai Jiao Tong University
Shanghai, China
yyjess@sjtu.edu.cn

***Abstract*—Cloud-native-based software development is in trend now. The end-users use the cloud services to save the local computation resources for other intensive tasks. Compiling is one of the vital required services. The *compiling-on-the-cloud* service like *CloudCompiling* or *CityCloud* requires the user to upload their source code for online compiling. However, the latest online compiling service can neither protect the users' source code privacy nor prove the integrity of the whole compiling process. To fill this gap, we designed COMURICE to provide a user-transparent, secure compiling service employing the trusted execution environment (TEE) to enforce security by blocking all the attempts in code or data theft during the compiling procedure.**

COMURICE leverages the hardware security feature of TEE to prevent the compiling process from malicious access and modification while encrypting the communication channel to protect the integrity and privacy of the source code. The challenges in realizing COMURICE lie in porting a fully functional compiler such as GCC or LLVM and designing an efficient compiling service to minimize the performance lag brought by confidential computing. According to the characteristics of the compiling process, it consists of several routines, pre-processing, compiling/obfuscation, and linking. The division of the routines requires multiple enclaves to run simultaneously.

In the experiment, we compare COMURICE's compiling service with nativeLLVM, SCONELLVM, and GrapheneLLVM. From a performance perspective, COMURICE pays a fair cost for security. Generally, a project compiling with COMURICE suffers 1-2 times more performance loss than nativeLLVM. Compared to other confidential compiling techniques like GrapheneLLVM or SCONELLVM, COMURICE is up to 20 times faster when compiling the same projects.

I. INTRODUCTION

With the development of cloud infrastructures and the growing requirement of cloud services, *compiling-on-the-cloud* has become a new trend. Enterprises like *CloudCompiling* [1] and *CityCloud* [2] provide the *compiling-on-the-cloud* service as the newly-raised solutions to the public. For an individual developer, the *compiling-on-the-cloud* offers the computing resources for the program compiling to ensure that the developer's local workstation is free to do other computing-intensive tasks. Besides, the *compiler-on-the-cloud* can provide the compiling environment in different versions for developers in need, which can also bring back convenience for the individual developers. For the Small-and-Medium-Sized-Enterprises (SMSEs), *compiling-on-the-cloud* is cheaper and easier to maintain compared to building the local Continuous Integration / Continuous Delivery (CI/CD) pipelines.

However, security is a significant concern during the *compiling-on-the-cloud* process. Individual developers rarely upload their source code to the cloud for Cloud Compiling, and SMSEs often build the CI/CD Pipelines locally during the development process instead of deploying the whole compiling environment on the Cloud Server. According to the survey, 95% of the cloud-native practitioners show security and privacy concerns about the cloud platform and their services [3]. Any malicious attempts on the cloud server-side, including process debugging or memory dump, would result in source code and other sensitive information leaks. The loss of the software developer's intellectual property and the hardcoded credentials in the source code would cause even wider attack surfaces [4].

Designing and implementing a secure, reliable, and practical cloud-based compiling/analyzing system should simultaneously meet the security needs of the Cloud Service Users (CSUs) and Cloud Service Providers (CSPs). On the one hand, the system should protect the integrity and privacy of the CSUs' code during the compiling and building process. Any source code leakage would trouble the CSUs a lot. On the other hand, the CSPs' intellectual property of the compiling/analyzing system should also be protected from data theft or dynamic debugging since the public server where the CSPs' deployed their services may be compromised during the run time.

A practice in secure compiling is to build the cloud-based service on Trusted Execution Environment (TEE) such as Intel SGX [5] or AMD SEV [6]. It is now the leading hardware-based technology in cloud confidential computing. For instance, in the most prevailing TEE solution, Intel SGX, software running in the enclave consists of a secure and

2693-8928/24/$31.00 ©2024 IEEE
DOI 10.1109/CSCloud62866.2024.00029

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on March 18,2026 at 08:13:43 UTC from IEEE Xplore. Restrictions apply.

an insecure component. Enclaves provide sensitive data and code another secure execution approach when facing the robust process debugging or physical attack on the cloud server, even running on a compromised operating system or hypervisor. CPU-assisted memory encryption on the secure components provides confidentiality to the user program but causes colossal performance loss. Those are the challenges we have to tackle when porting complex software, such as a compiling service, into the enclaves. Leveraging the security feature of the enclave, we could design and implement an Intel SGX supported cloud compiling service by porting and remodeling a prevailing compiling toolchain LLVM into the enclave.

We implement a secure cloud compiling/obfuscation service named COMURICE, which tackles the challenges by porting the full-function compiling system LLVM to the enclave. Based on the LLVM compiling framework, this multi-language supporting modern compiler provides confidentiality, integrity, and privacy for CSU's source code.

The biggest challenge for implementing COMURICE is to port a compiling service into the enclave. We choose LLVM, a large-scale compiling framework written in C++ with a collection of reusable compilers and toolchains. Porting methods such as SCONE [7] and Graphene-SGX [8] didn't optimize for the characteristics of the compiling service, so they introduce significant performance loss. Also, they have introduced a larger Trust Computing Base (TCB) than what is necessary. We designed a library wrapper tool only to provide the required interfaces. With this design, we can port a moderately large-scale software system into an enclave with a rather small and flexible size TCB.

The paper makes the following contributions:

- We design an efficient automated modernization method to port LLVM to the enclave while requiring a small TCB. By implementing the necessary C++ interfaces, our modernization method can support C++ programs on the library interface level. Our design of the Shim Layer can be leveraged when porting other programs into SGX.
- We design and implement a cloud-based compiling service framework COMURICE guaranteeing confidentiality and integrity of the source code and the intermediate files. It's an extensible building tool that supports various front-ends, including Clang, FORTRAN, Rust, etc. Also, it can output the hardened binary files by obfuscation.
- We make tailored optimization to COMURICE by carefully taking the program into account that frequently startup application is more sensitive to the initialization process overhead. Thus COMURICE achieves better performance than other modernization methods in our dataset.

II. Background

Intel Software Guard Extensions(SGX) [9] is a hardware-enforced security technology, allowing users to create the user-mode trusted execution environment. Txhe central concept of SGX is the enclave that contains the code and security-sensitive data to resist attacks from other processes, hypervisors, OS, firmware, and even hardware. Generally speaking, SGX's protection mechanisms are intended to ensure the privacy and integrity of the computation running inside the enclave.

SGX SDK library. Since the program inside the enclave cannot call glibc functions directly, SGX SDK provides a self-contained LibC-like implementation called `sgx_tstdc` library to facilitate the developing process. `sgx_tstdc` contains many `glibc` routines implementation, but it still does not suffice to port LLVM into the enclave. The developer either has to implement a complete LibOS (as in Graphene-SGX) or forward the function calls to the system code outside the enclave.

ECall/OCall. Program running inside enclave may need access to system calls that are unavailable in the enclave. Therefore the program does a context switch to invoke the untrusted routines, which will forward the system call requests to the operating system. Context switch from enclave to non-enclave is called ECall, while the switch from non-enclave to enclave is called OCall. Intel provides special instructions to do the context switch between the enclave and non-enclave programs. SGX SDK implements the corresponding ECall and OCall functions by wrapping the instructions. However, frequent ECall and OCall can bring lots of context switches that incur high-performance overhead. During the programming, the developer should define all the ECall/OCall methods in the *.edl* files. The SGX SDK would parse the *.edl* file into *.cpp/.h* files before the final compiling.

III. An Illustrative Example

Figure 1 shows an illustrative example of using COMURICE to support confidential obfuscation service. The CIP launches COMURICE at the very beginning, which would then launch several enclaves. The number of enclaves needed can be determined by CSP, who would take the number of obfuscation tasks and the workloads of the public server into consideration previously. After COMURICE service is ready, the CSU would check the integrity of the confidential obfuscation service by using remote attestation supported by Intel. Intel Attestation Service would offer relevant reports to prove the service is as it is supposed to be. Then, the CSU would upload the source code to COMURICEor input the address of the code repository. The message communicated between source code and COMURICE has been encrypted through a pre-built TLS channel.

COMURICE would handle the input file under confidential circumstances by leveraging the Intel SGX technique. In COMURICE, three kinds of SGX enclaves have been created previously, including one Daemon, multiple Obfuscation Enclaves (OEs), and multiple Linking Enclaves (LEs). The compiling step is executed inside the OE. During the obfuscation, Daemon would automatically distribute the source file to the corresponding enclave. Each enclave would handle the source file beginning with TLS decryption. After compiling

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on March 18,2026 at 08:13:43 UTC from IEEE Xplore. Restrictions apply.

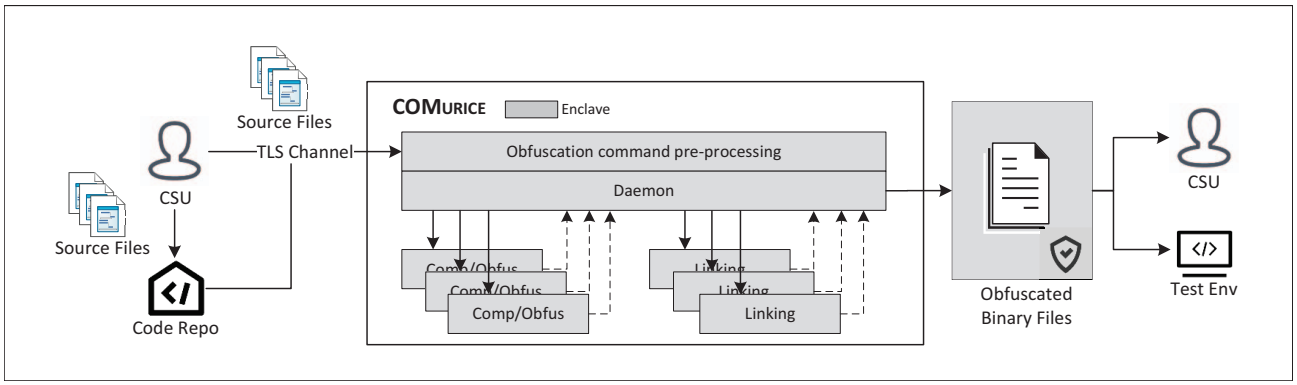


Fig. 1. The compiling workflow of COMURICE.

and obfuscation, the target file would be sent to LE and linked with link libraries. The generated file is an obfuscated binary. The result would be sent back to CSU or directly sent to the test environment on demand.

IV. INNER DESIGN

This section would describe the structure of COMURICE's enclave at the library level, including Daemon, OE, and LE.

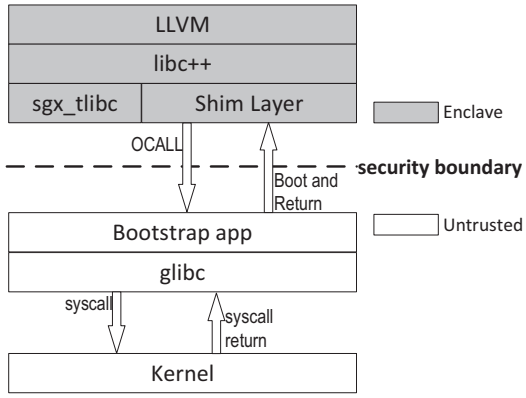


Fig. 2. Inner design of COMURICE.

COMURICE's design is shown as Figure 2. The whole system consists of two parts, which are the trusted and the untrusted, respectively. We port the whole LLVM source code into the trusted part without any modification. The `sgx_tlibc`, `libc++`, as well as the Shim Layer, provide the complete interfaces to LLVM. The `libc++` is the standard C++ library that offers C++ interfaces, while the `sgx_tlibc` is a library of tailored C interfaces offered by SGX SDK running inside the enclave. The Shim Layer, which contains several stub functions, aims to deal with unsupported system calls inside the enclave. Each stub function would call a specific OCall method in the untrusted program space. Apart from offering the interfaces, the Shim Layer has two main effects. The first one is that it can protect the behavior of data access during compiling from untrusted OSes. In other words, the Shim Layer would do the encryption/decryption while reading or writing the sensitive data. The second effect is that the Shim Layer can protect the whole compiling system from malicious attacks under the untrusted OS by doing the boundary check after the stub function receives the untrusted data. Also, COMURICE has a special design for multi-process support. The OCall methods are implemented in the untrusted region and depend on the OS's standard C library. The whole COMURICE runs in the host OS over the platform OS kernel.

Bootstrap clang. Clang is the frontend of LLVM. In COMURICE, clang is the entrance of the enclave program. On the REE side, a user level `main` function would call `ecall_compile` to enter the enclave. On the TEE side, the `ecall_compile` would invoke clang in the enclave. Depending on functionality, the invoked enclave can either be Daemon, OE, or LE. Besides, `ecall_compile` would process and transmit the parameters, whose integrity can be guaranteed by TEE.

LibC/LibC++ Support. As programmed in C/C++, the native LLVM relies on `stdlibc++` and `glibc` to provide library and syscall support. However, these two libraries cannot be used inside the SGX enclave due to the fact that the SGX does not support syscall. Intel SGX SDK provides a tailored C library called `sgx_tlibc`, which shares the same function symbol as the standard C library function and gives a part of the standard C interface support. This cannot fill the whole gap. The straightforward idea is to rewrite the missing interfaces inside the enclave with the help of the OCall method offered by SGX SDK. Unfortunately, the latest version of SGX SDK does not support the C++ OCall method. Moreover, implementing all missing C++ interfaces would bring back unaffordable workload as well as security threats. Therefore, we port the whole *libc++* into the enclave during the linking process to offer the C++ interface support.

Shim Layer Design. The Shim Layer is designed to make LLVM compatible with SGX. It is the boundary between the TEE and the REE. It contains several stub functions

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on March 18,2026 at 08:13:43 UTC from IEEE Xplore. Restrictions apply.

which can offer interfaces referenced by LLVM and LibC++. These interfaces would not only provide basic functionalities, but also some advanced features, like multi-process support, encryption/decryption, and return value check to defend Iago Attack. In general, COMURICE provides 115 stub functions in total. Three of them contain encryption methods, and five of them contain decryption methods. Despite this, 26 stub functions have *errno* as the parameter, which can guide the compiler to find the correct trace of library dependency. The rest of the stub functions are designed in a unified format.

V. EVALUATION

We have evaluated COMURICE on a set of benchmarks. Our evaluation is designed to answer two research questions:

- **RQ1**. Does COMURICE indeed prevent source code leakage in every stage of COMURICE 's workflow?
- **RQ2**. Can COMURICE achieve competitive performance on confidential compiling and obfuscation tasks without losing correctness?

A. Setup

1) Techique: We compare COMURICE with three different techniques: NativeLLVM, GrapheneLLVM, and SCONELLVM. As there is no off-the-shelf project, we port the LLVM binary into the LibOSes as the comparative objects.

- **NativeLLVM**. This is the baseline. To eliminate the loss, we use C++ static library to build the LLVM project (or clang frontend).
- **GrapheneLLVM**. GrapheneLLVM is a LLVM-in-enclave, LibOS-based solution. It uses Graphene-SGX (a fully-featured LibOS) as the underlying system to enable LLVM to run inside the SGX enclave. Every time a compiling or obfuscation command is loading, Graphene-SGX would boot the `pal-loader` and create an enclave. After this, the LLVM program would be started inside the enclave to finish the task.
- **SCONELLVM**. Same as GrapheneLLVM, SCONE-LLVM is also a LibOS-based solution of LLVM-in-enclave. However, the public version of SCONE does not have multi-process support, which is widely used in LLVM. In the experiment, we manually separate the compiling and linking process and add the time cost as the result.

2) Dataset: Table I shows an overview of the selected projects as a dataset for our evaluation. These are the projects written in C/C++, including `7-zip` [10], `CPython` [11], `Crypto++` [12], `FFmpeg` [13], `ld` [14], `Lua` [15], `musl-libc` [16], `OpenLibm` [17], `libtiff` [18] and `wolfSSL` [19], which are 2 interpreters, 4 third-party libraries, and 4 tools. Also, these projects are various from 1.7 MB to 91.8 MB in size, which can somehow demonstrate the efficiency and robustness of the confidential compilers.

3) Configuration: Our evaluation is conducted on a host equipped with Intel Core i5-9400, Ubuntu 18.04. Also, we set the Intel SGX SDK in version 2.13 in our platform with its corresponding SGX driver. All the confidential LLVM techniques are built with LLVM 3.4 with the obfuscation pass.

B. Confidentiality

We would like to answer the **RQ1** in this section from two aspects. First, we must ensure that COMURICE's TCB size is small enough to avoid potential attacks. Secondly, we should prove that COMURICE can protect the CSU's source code against known attacks.

1) TCB Size: We compared the TCB size of COMURICE with GrapheneLLVM and SCONELLVM. The result shows that COMURICE reduces about 30 kLoCs than GrapheneLLVM, and more than 70 kLoCs than SCONELLVM. This can somehow reduce the attack surface of confidential compiling service.

One main reason is that GrapheneLLVM and SCONELLVM are built with SGX libOSes, which contains multiple functions which would not be used by LLVM program.

2) Security Features: We discuss the security features of COMURICE from 4 aspects.

Runtime Confidentiality and Integrity. This feature is guaranteed by the Intel SGX hardware mechanism. The SGX can protect the program running inside the enclave even if the host server is compromised by a malicious attacker. The target source code files are encrypted when on REE side and would be decrypted on the TEE side. This would not leak CSU's sensitive information.

Defense Against Iago Attack. The data tampering attack, especially the Iago attack [20], would happen when the host OS is compromised by an attacker who has access to the server's operating system. The attacker would change the *return values* or *pointer values* of OCall functions in order to hijack the control flow and finally read or write the sensitive memory. In COMURICE, we do the boundary check during the OCall methods. The *return values* would always be in a limited scope and would not result in severe mistakes if modified into limited values. The *pointer values* would also be checked if potential data overflow would happen. Also, the EDL files would limit the content inside the pointers.

Defense Against the MITM Attack. When using the online compiling service, the security feature on the communication channel should also be guaranteed. Since the attacker can physically access the server, he can perform the Replay Attack or the Man-in-the-Middle (MITM) Attack. After the Remote Attestation, the CSUs would confirm that the compiling service is genuinely running inside the enclave. Also, a communication key would be generated based on the Diffie-Hellman Key Exchange process. This key could be leveraged to do the end-to-end encryption and thus prevent attacks on the communication channel.

File System Security. Intel SGX does not support the system call inside the enclave. When performing I/O operations, the program inside the enclave would call the OCall method and transfer the content to the untrusted memory. Therefore, in COMURICE, all the read/write methods encapsulated in the shim layer would contain the data encryption/decryption

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on March 18,2026 at 08:13:43 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SECURE COMPILERS' PERFORMANCE ON DIFFERENT PROJECTS

Projects	Native	COMurice	Graphene	SCONE	Language	# of Files	Size	Functionality
7-zip	0m38s	1m04s	19m50s	9m59s	C/C++	420	4.1M	Compression tool
CPython	0m41s	1m30s	13m56s	9m39s	C/C++	333	91.8M	Python interpreter
Crypto++	2m10s	10m44s	55m30s	22m29s	C++	270	22.4M	Crypto library
FFmpeg	5m39s	17m27s	179m08s	124m03s	C/C++	2793	71.5M	Video decoder
ld	0m14s	0m16s	3m23s	×	C	597	21.7M	Linker
Lua	0m05s	0m08s	3m10s	1m11s	C	40	1.7M	Lua interpreter
musl-libc	1m04s	30m59s	232m31s	×	C	1580	4.3M	C static library
OpenLibm	0m05s	1m20s	22m59s	×	C/C++	353	18.1M	Mathematical library
tiff	0m14s	0m31s	12m46s	×	C/C++	107	7.9M	Tag Image File Format
wolfSSL	0m15s	0m29s	5m47s	×	C/C++	285	57.3M	A SSL/TLS library

Graphene and SCONE represent GrapheneLLVM and SCONELLVM, respectively. × means that the compiler could not finish the compiling task. # of files only include the number of C and C++ files.

operation. All the data originating from the enclave would be stored on the disk in ciphertext to prevent illegal data access from the attacker.

C. Performance

We would like to answer the **RQ2** in this section. The core components of confidential integration tasks are the confidential compiling and obfuscation steps, and these steps are the majority source of performance overhead. Benchmarks running on **GrapheneLLVM**, **SCONELLVM**, and COMURICE are in general slower than those on **NativeLLVM**. Each project, when compiled by confidential compilers, needs multiple enclaves. The creation or termination of the enclaves, and the ECall/OCall methods, would bring back large performance overhead. In this section, we would compare the execution time of compiling projects on different techniques in various execution modes.

We compile these projects in `cmake & make` sequence or `configure & make` sequence. During the `make` step, the confidential compiler would execute most of the commands except the linking. That is because SCONE LibOS does not support the startup of a new process in the enclave at runtime. For fairness, we removed the linking process from the `Makefiles`. Some of the projects still cannot be compiled under the **SCONELLVM** environment since they contain assembly code files. The crash would happen when **SCONELLVM** handles these files due to the need for new child process creation.

1) Single-core compiling: Table I shows the execution time of running benchmarks in single-core compiling modes. The so-called single-core compiling means only one source code file would be compiled at any time. This can most intuitively reflect the performance overhead of different techniques.

Compared to the other two confidential compiler techniques, COMURICE only takes 5% to 20% of the time cost to finish the same task. For example, when compiling `FFmpeg`, which is a project with 2793 files to be compiled, **GrapheneLLVM** and **SCONELLVM** would take more than two hours, while COMURICE takes only 17 and a half minutes. The unbearable performance overhead when using **GrapheneLLVM** and **SCONELLVM** comes from the frequent creation and termination of the enclave, which is unnecessary. Every time a compiling command is loading, the **GrapheneLLVM** or **SCONELLVM** would launch a new enclave and check the integration. After the operation is finished, the Daemon would terminate the enclave. In COMURICE , the unnecessary creation and termination steps have been cut out, which largely reduce the time cost of confidential compiling.

Compared to the **NativeLLVM**, COMURICE would take more than 2x time cost when compiling the same project. When compiling those projects which would take 3x or more extra time cost, COMURICE's enclave would be crashed because of the memory exhausted. In our experiment, we could only reboot the enclave and continue the task. Undoubtedly, this would bring back large time costs. The core reason for memory exhaustion is the unstrict memory management of LLVM 3.4. When finishing a compiling command, some useless data has been stored in the heap where it is supposed to be freed. When there is no need to reboot the enclave, COMURICE would only take 100% more time cost than **NativeLLVM** when doing confidential compiling.

2) Multi-core compiling: In order to build projects in an efficient way, the developer always `make` the projects in a multi-process way. To be specific, it is to add "-j" option during the `make` step. COMURICE maintains this feature in the confidential compiling. Figure 3 shows the comparison of the performance among four compiler techniques. We choose `lua` as the target to evaluate the multi-core compiling support of the three confidential compiler techniques. The result shows that COMURICE would about 2x time cost to finish the tasks, while **GrapheneLLVM** and **SCONELLVM** would spend 25x to 100x time cost to do the same compiling.

As shown in figure 3, **GrapheneLLVM** and **SCONELLVM** has performance bottleneck, for the time cost of the "j3" and "j4" is nearly the same. Intuitively, this is because the creation and termination of the enclaves are more difficult to parallelize than confidential compiling operations. COMURICE bypasses the bottleneck in a smarter way. When building a project in a multi-process way, COMURICE would previously launch several compiling enclaves and linking enclaves. *Daemon* would distribute the compiling tasks to the different enclaves. There is no need to launch a new enclave during the compiling, which greatly lowers the time cost. The result shows that COMURICE

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on March 18,2026 at 08:13:43 UTC from IEEE Xplore. Restrictions apply.

has better support for parallel confidential compiling than the other two confidential compiler techniques.

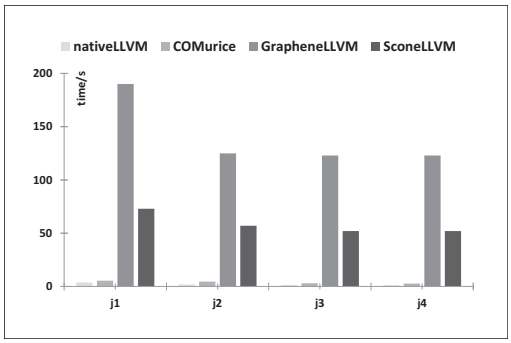


Fig. 3. Compiling **lua** with multiple cores

VI. Discussion

Incubated on LLVM, COMURICE leverages the advantages of LLVM, such as IR support and scalability in front-ends and back-ends. It could support multiple programming languages if provided with a proper front-end. We only port the Clang, which is the C/C++ front-end of LLVM. Other front-ends can also be supported under the same porting method. Also, it can compile the source code into different target binaries on aimed for different architecture. LLVM also has WebAssembly (WASM) support. COMURICE can build a whole confidential computing process from compiling to execution combining to other existing work like AccTEE [1] or CHANCEL [21], which runs a WASM runtime inside the Enclave. Moreover, LLVM has some extra passes, which can do the analysis and obfuscation work on IR. These passes can also be integrated into this work.

Our COMURICE still has some issues with heap allocation efficiency inside the enclave. This may be brought in by LLVM's implementation, for it could not free all the heap space allocated by itself properly. If we can tackle this heap allocation issue, the enclave would not crash during the long execution, and there is no need to reboot the enclave and thus can achieve higher performance.

Moreover, our method of porting large-scale projects into the Intel SGX can also be referenced and used in other modernizing jobs to reduce the time of enclave creation and termination.

VII. Conclusion

Trusted execution environment technology has assisted a myriad of applications in achieving the goal of confidential computing on real machines. COMURICE brings into sight the SGX-based implementation on a secure cloud compiling service. COMURICE has offered a secure online compiling experience by implementing the most widely utilized compiling toolchain, LLVM, and an automated and highly efficient mechanism in porting a complex software system into SGX enclave to two well-known poring methods SCONE and Graphene-SGX.

References

[1] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "Acctee: A webassembly-based two-way sandbox for trusted resource accounting," in Proceedings of the 20th International Middleware Conference, ser. Middleware '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 123–135. [Online]. Available: https://doi.org/10.1145/3361525.3361541

[2] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 693–707.

[3] C. N. C. Foundation. (2019) Cncf annual report 2019. [Online]. Available: https://www.cncf.io/cncf-annual-report-2019/

[4] M. Meli, M. R. McNiece, and B. Reaves. (2019) How bad can it git? characterizing secret leakage in public github repositories.

[5] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy, vol. 13. ACM New York, NY, USA, 2013, p. 7.

[6] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," White paper, 2016.

[7] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "Scone: Secure linux containers with intel sgx," in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, ser. OSDI'16. USA: USENIX Association, 2016, p. 689–703.

[8] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in 2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA, 2017, pp. 645–658.

[9] Intel, "Intel® software guard extensions," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html

[10] I. Pavlov, "p7zip," 2021. [Online]. Available: https://www.7-zip.org/t

[11] Python, "Cpython," 2021, accessed on 04/20/2021, Commit 0xb66fbd1. [Online]. Available: https://github.com/python/cpython.git

[12] C. Community. (2021) Cryptopp. [Online]. Available: https://cryptopp.com/

[13] F. Community. (2021) Ffmpeg. [Online]. Available: https://git.ffmpeg.org/ffmpeg.git

[14] Python. (2021) binutil gdb ld. [Online]. Available: https://gitlab.redox-os.org/redox-os/binutils-gdb

[15] L. Community. (2021) Lua. [Online]. Available: https://github.com/lua/lua.git

[16] musl community. (2021) musl-libc. [Online]. Available: https://musl.libc.org/

[17] the Julia Project. (2021) Openlibm. [Online]. Available: https://github.com/JuliaMath/openlibm.git

[18] anonymous. (2021) libtiff. [Online]. Available: https://github.com/LuaDist/libtiff.git

[19] wolfSSL Community. (2021) wolfssl. [Online]. Available: https://github.com/wolfSSL/wolfssl.git

[20] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2013, p. 253–264.

[21] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "Chancel: efficient multi-client isolation under adversarial programs," in Annual Network and Distributed System Security Symposium (NDSS), 2021.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on March 18,2026 at 08:13:43 UTC from IEEE Xplore. Restrictions apply.