# Shining Light on the Inter-procedural Code Obfuscation: Keep Pace with Progress in Binary Diffing*

PEIHUA ZHANG, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and WeChat, Tencent, Beijing, China
CHENGGANG WU†‡, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China, University of the Chinese Academy of Sciences, Beijing, China, and Zhongguancun Laboratory, Beijing, China
HANZHI HU†, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and Institute of Computing Technology Chinese Academy of Sciences, Beijing, China
LICHEN JIA, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, China
MINGFAN PENG†, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, China
JIALI XU, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China
MENGYAO XIE, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China
YUANMING LAI†, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, China
YAN KANG†, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, China
ZHE WANG†‡§, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China, University of the Chinese Academy of Sciences, Beijing, China, and Zhongguancun Laboratory, Beijing, China



*Extension of conference paper: An earlier version was presented at CGO 2023 [69]. We do further study and extend the conference paper in several directions: 1. Combined with the new observation of binary diffing works that pay more and more attention to the control flow semantics, we propose a new obfuscation primitive called *hidden*, which utilizes the exception-handling mechanism to conceal the control flow (Section §3.4). 2. We provide guidelines on selecting and combining the primitives to achieve a higher level of obfuscation while minimizing runtime overhead (Section §3.5). 3. We introduced 11 binary diffing works including intra-procedural and inter-procedural diffing. Our results demonstrate that the enhanced approach improves performance and exhibits higher efficiency (Section §5.2). Additionally, we conducted experiments to collect internal information, providing further evidence of the effectiveness (Section §5.5). 4. To gain insights into the impact of actively utilizing compiler optimization in obfuscation, we compare KHAOS with BinTuner [52]. This comparative study reveals distinguishing factors and presents new observations (Section §5.4).

†also with University of Chinese Academy of Sciences.
‡also with Zhongguancun Laboratory.
§is the corresponding author.

Authors' Contact Information: Peihua Zhang, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and WeChat, Tencent, Beijing, Beijing, China; e-mail: zhangpeihua@ict.ac.cn; Chenggang Wu, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, Beijing, China and Zhongguancun


ACM 1544-3973/2024/10-ART
https://doi.org/10.1145/3701992

Software obfuscation techniques have lost their effectiveness due to the rapid development of binary diffing techniques, which can achieve accurate function matching and identification. In this paper, we propose a new inter-procedural code obfuscation mechanism KHAOS[1], which moves the code across functions to obfuscate the function by using compilation optimizations. Three obfuscation primitives are proposed to separate, aggregate, and hide the function. They can be combined to enhance the obfuscation effect further. This paper also reveals distinguishing factors on obfuscation and compiler optimization and presents novel observations to gain insights into the impact of actively utilizing compiler optimization in obfuscation. A prototype of KHAOS is implemented and evaluated on a large number of real-world programs. Experimental results show that KHAOS outperforms existing code obfuscations and can significantly reduce the accuracy rates of six state-of-the-art binary diffing techniques with lower runtime overhead.

CCS Concepts: • **Security and privacy → Software and application security**.

Additional Key Words and Phrases: Software Protection, Obfuscation, Binary Diffing

## 1 Introduction

Embedded devices, including wearables, traffic lights, and autonomous driving vision sensors, have become ubiquitous in modern life. Unfortunately, their disclosure of vulnerabilities has also been increasing, leading to a corresponding rise in attacks. Exploiting a vulnerability in them can have severe consequences, ranging from the collapse of critical network infrastructure to life-threatening situations in the case of medical devices like pacemakers.

The vulnerabilities can be attributed not only to the writing of flawed code but also to the reuse of vulnerable code. This practice significantly contributes to the widespread presence of vulnerabilities in embedded devices [10, 41, 47]. However, addressing these vulnerabilities is challenging due to fragmentary issues. The fast-paced replacement results in similar code being present across multiple versions of various products, making timely patching difficult [62].

The above problem favors attackers to detect existing vulnerabilities instead of exploring 0-day vulnerabilities laboriously. Since most embedded device software is not open source, attackers usually utilize the binary diffing techniques [5, 8, 13, 16, 17, 22, 28, 38, 50, 59, 67, 71, 72] to locate the reused vulnerable code by comparing the binary with the third-party code. With the introduction of deep learning, binary diffing techniques have made great progress. For example, David et al. [14] searched for common vulnerabilities in common devices, and were able to locate 373 existing vulnerabilities. The method designed by Feng et al. [21] can achieve a 10,000-level diffing in less than 1 second. In this scenario, it is particularly important to protect the software security of embedded devices by countering the binary diffing technique.

Software obfuscation techniques [3, 9, 26, 32, 60, 64] can transform the code to change the binary. They could be used against binary diffing techniques, preventing attackers from locating existing vulnerabilities. However, recent research has shown they are no longer effective against the SOTA binary diffing techniques [16, 40, 45, 61].

[1]https://github.com/Khaos2022/Khaos-master

Laboratory, Beijing, China; e-mail: wucg@ict.ac.cn; Hanzhi Hu, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and Institute of Computing Technology Chinese Academy of Sciences, Beijing, Beijing, China; e-mail: huhanzhi22s@ict.ac.cn; Lichen Jia, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, China; e-mail: lcjia457@gmail.com; Mingfan Peng, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, China; e-mail: pengmingfan20g@ict.ac.cn; Jiali Xu, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China; e-mail: zoecur3@gmail.com; Mengyao Xie, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China; e-mail: xiemengyao@ict.ac.cn; Yuanming Lai, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, Beijing, China; e-mail: laiyuanming@ict.ac.cn; Yan Kang, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, Beijing, China; e-mail: kangyan@ict.ac.cn; Zhe Wang, SKLP, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China and University of the Chinese Academy of Sciences, Beijing, China and Zhongguancun Laboratory, Beijing, China; e-mail: wangzhe12@ict.ac.cn.

The main reason is that most software obfuscation techniques focus on intra-procedural obfuscation, which does not fundamentally change the semantics of functions, while binary diffing techniques can more and more accurately extract features within functions to obtain their semantics. Based on the above observations, we argue that *inter-procedural code obfuscation should be emphasized due to its ability to change function semantics.*

To this end, we propose an inter-procedural obfuscation technique, KHAOS, which moves the code across functions and utilizes the compiler's optimizations to transform (obfuscate) the code. The core idea of KHAOS is that *once the code is restructured among functions, the generated binary code after compilation optimizations can be very different.* To achieve the inter-procedural code obfuscation, KHAOS changes function code across functions by separating a function into sub-functions, aggregating functions into one, and hiding control flow inside/outside functions.

It is non-trivial to transform arbitrary functions due to the challenges posed by performance, correctness, and obfuscation effects. For example, 1) To balance the obfuscation effect with the performance, choosing which code blocks within a function to be separated is a problem; 2) Rebuilding all control flow and data flow among functions after aggregation is difficult; 3) Hiding control flows effectively without bringing too much overhead is difficult. To address these challenges, three obfuscation primitives are proposed in KHAOS— *fission*, *fusion*, and *hidden*.

The fission partitions the code region to a sub-function with the dominator tree as the granularity and also combines the static cold/hot code analysis technique to lower performance overhead. Since the define-use relationships of variables are changed from within a function to cross functions, it needs to rebuild the data flow by passing parameters. To minimize the performance degradation caused by parameter passing, we also propose a data-flow reduction mechanism to reduce the number of parameters of the sub-functions. The control flow is also rebuilt by inserting the function calls that call to sub-functions and encoding the return values in the sub-function.

The fusion aggregates functions with compatible return values and merges their parameter lists. To avoid the inefficient way of stack passing parameters, we propose a compression mechanism to reduce the number of parameters. To rebuild the control flow completely, we propose a tagged pointer mechanism, which attaches control bits on function pointers to decide the executed code when the aggregated functions are called indirectly. To further improve the obfuscation effect, the deep fusion method is proposed to merge innocuous basic blocks, whose execution does not affect the global memory state, from different functions together within the aggregated function.

The hidden utilizes the exception-handling mechanism to hide control flows. The exception-handling mechanism is a key feature of C++ that developers can capture actively or passively generated exceptions in the try block by writing the catch statements. It requires the cooperation of the system ABI and the exception-handling runtime library. They select the appropriate handling code to execute by searching the exception-handling information recorded in the binary. We hide the program's control flow by transforming direct jump into a throw-and-catch relationship, which becomes a lookup-based indirect control flow. To further hide indirect control flow, we propose a *code dealer* to randomly send the hidden code to other functions at compile-time. We also propose complementary methods for C programs.

KHAOS was implemented based on the LLVM framework. The experimental evaluations were conducted on the Linux/X86_64 platform using SPEC CPU 2006 & 2017 C/C++ programs, CoreUtils, and 5 common embedded device software containing vulnerabilities. Eleven SOTA binary diffing tools [16–18, 23, 43, 49, 56, 59, 63, 65, 73] were used to evaluate the effectiveness of KHAOS. The results show that KHAOS is not only effective but also efficient: the effectiveness experiments show that the accuracy of these binary diffing tools was reduced to be less than 19%, and the ranking of the vulnerable functions decreased significantly; the performance experiments show that KHAOS incurs less than 7% overhead on average. In summary, our contributions can be outlined as follows:

- **An inter-procedural obfuscation mechanism.** We recognize the importance of inter-procedural obfuscation in countering binary diffing techniques. To address this need, we introduce an obfuscation mechanism called KHAOS, which obfuscates code across functions.
- **The three complementary primitives.** Within KHAOS, we propose three obfuscation primitives to facilitate the movement and concealment. These primitives include fission, which divides a function into multiple functions; fusion, which combines multiple functions into one; and hidden, which leverages the exception mechanism to hide the control flow.
- **New insights from implementation and evaluation.** We have developed and evaluated a prototype of KHAOS, and the results show that it outperforms the existing obfuscators against the SOTA binary diffing techniques. Our study suggests that binary diffing techniques should focus more on the inter-procedural code features.
- **New observation on compiler optimization regarding obfuscation.** Code obfuscation needs to combat compiler optimizations so as not to generate the same binary code, while the compiler itself can achieve a limited obfuscation effect due to its pattern-relative nature, our research suggests that a more effective way is to obfuscate the program in a compiler-comply manner, leveraging the compiler to enhance the obfuscation effect.

## 2 Background and Motivation

### 2.1 Binary Diffing

Binary diffing is a technique for visualizing and identifying differences between binaries. It can quantitatively measure the differences and give matching results at predefined granularity (e.g., function). It has been widely used in software vulnerability search, security patch analysis, malware detection, code clone detection, etc. There are two kinds of binary diffing works [17], traditional approaches, and learning-based approaches.

- Traditional methods match binaries by counting specific statistical information, such as the aforementioned BinDiff [73]. To deal with cross-compiler/architecture scenarios, many works [8, 28, 40, 45, 50, 61, 67] try to extract semantic-level features as the code snippet's identity, such as using I/O syntax [8] to describe a basic block.
- Recently, many works tried to adopt machine learning to perform the diffing [16, 17, 22, 38, 59, 68, 72]. For example, Asm2Vec [16] regards assembly language as a special language, abstracts each element (opcode, operands) in the program as tokens in the natural language, and generates the representation of each token through training and clustering.

### 2.2 Software Obfuscation

Software obfuscation transforms the program without changing its functionality to make it hard to analyze. There is an arms race between software obfuscation and binary diffing. Software obfuscation does not want binary diffing techniques to match un-obfuscated with obfuscated code successfully, and vice versa. There have been various techniques proposed in software obfuscation, which can be classified into data obfuscation, static code rewriting, and dynamic code rewriting [54].

Data obfuscation techniques transform the data format to prevent it from matching. For example, variables like arrays can be shuffled and split into several parts, which can be used to hide the secret. Since most binary diffing techniques utilize the features of the code, obfuscating data is less effective, and we leave it as an orthogonal technique.

Various dynamic code rewriting approaches follow the concept of packing [46]. However, these techniques are easy to automatically unpack [48] or be memory-dumped [58], which would lose the effect of obfuscation. Code

virtualization is another popular obfuscation technique [34]. It translates code into specific representations and then uses an engine to interpret them at runtime. This technique sacrifices much performance (10x slowdown [34]) in exchange for a more powerful obfuscation. Therefore, the dynamic code rewriting technique is not suitable for fighting against binary diffing due to less effectiveness or too much overhead.

In contrast, static code rewriting is a promising technique against binary diffing. It modifies program code during obfuscation without further runtime modifications, which is similar to compiler optimization. Researchers have proposed many techniques for static code rewriting. For ease of introduction, we categorize them by obfuscation granularity:

**Instruction level:** Instruction substitution [9, 32] replaces the original instruction with equivalent instruction(s), such as replacing an "`add`" with two "`sub`". O-LLVM [32] designed 10 different strategies for arithmetic and logical operations. To increase the complexity of the conditional branch, opaque predicate techniques [9, 32, 64] were proposed. They add permanent true or false (e.g., $x^2$ != -1) conditions that do not affect the original control flow, which is frequently used against analytical techniques such as symbolic execution.

**Basic block level:** Bogus control flow [9, 32] inserts dead code into the original control flow and often utilizes opaque predicates to prevent these codes from being optimized away and executed, thereby ensuring the original functionality of the program.

**Function level:** Control flow flattening [9, 32] converts the control flow of the function into the "`switch-case`" form, which is hard to analyze, and maintains the original jump relationship by controlling the values of the cases. To prevent being degraded back to the original control flow, the "`case`" relationship is also obfuscated (encrypted).

In terms of binary diffing, compared with static code obfuscation, dynamic code rewriting(e.g., code virtualization) may achieve a similar or even stronger adversarial effect. However, it often requires an additional interpreter program and brings high runtime overhead, thus resulting in limited scenarios. Packing, as a special kind of obfuscation, can be automatically unpacked or memory-dumped[58], which would lose the effect of obfuscation. In comparison, static code obfuscation is more natural and effect-preserved.

### 2.3 Motivation

Since code obfuscation is a common source contributing to binary code differences, testing the resilience against obfuscation has become a common evaluation step for binary diffing [16, 40, 61]. From their evaluation, obfuscation techniques with intra-procedural granularity(e.g., statement, basic block, function) have failed in this arm race [16]. The reason is that intra-procedural obfuscation techniques do not fundamentally change the semantics of each function. With the continuous improvement of binary diffing techniques in feature extraction and binary code representation, especially with the application of deep learning, their ability to capture the semantics is becoming increasingly more robust so the effect of intra-procedural obfuscation is gradually weakening.

Therefore, we argue that *inter-procedural code obfuscations should be emphasized due to their ability to change function semantics which is the key to defeating binary diffing*. Our thinking is also hinted by the literature published from both the offensive and defensive sides: 1) most of the binary diffing works have discussed the issues of function inline [2, 5, 8, 13, 15, 16, 19–21, 27–30, 40, 66], and many of them [2, 14, 15, 19–21, 28–30, 66] admitted that it would affect the accuracy of diffing; 2) BinTuner [52] found the function inline could reduce the diffing accuracy by approximately 10%.

## 3 Our Solution: KHAOS

### 3.1 Overview

To achieve the inter-procedural obfuscation, KHAOS changes the amount of code within a function by moving code across functions first and then utilizes the compiler's optimizations to transform (obfuscate) the code. The

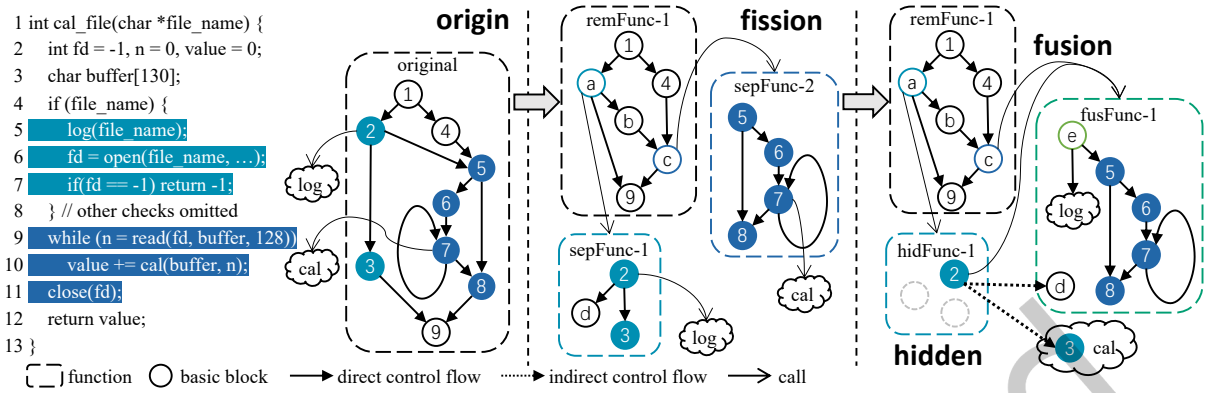


Fig. 1. An example of obfuscating a function by using KHAOS.

fission primitive separates a function into multiple sub-functions thus making the function thinner. The fusion primitive aggregates functions into one thus making the function fatter. The hidden primitive hides the function's control flow and call relationship.

For the convenience of discussion, we denote a function before the transformation as an **oriFunc**, and the new function formed after the fusion as the **fusFunc**. The new function formed by the separated code during the fission is denoted as the **sepFunc**, and the function formed by the remaining code is denoted as the **remFunc**. The function processed after the hidden as the **hidFunc**. In Fig. 1, an example is presented to illustrate how KHAOS performs transformations on the `cal_file()`, which determines the number of special characters in a given file. It begins by examining the file name and opening the file (lines 4-7), followed by reading its content and counting the occurrences of special characters (lines 9-11).

Fission split the function into three functions: `sepFunc-1` contains basic block ② and ③, and `sepFunc-2` contains basic block ⑤ to ⑧. Trampolines (ⓐⓑⓒ) are inserted in `remFunc-1` to ensure correctness. Basic block (ⓓ) is utilized to return different values for `sepFunc-1` (detailed in §3.2).

Fusion merges the `log()` function and `sepFunc-2` into `fusFunc-1`. An entry basic block (ⓔ) is introduced to select the aggregated code blocks. To enable deeper code aggregation, we propose a deep fusion method that combines basic blocks from distinct function bodies. Once `fusFunc-1` is generated, the fusion process adjusts the references of `log()` and `sepFunc-2` to `fusFunc-1`.

To address the challenge of "stubborn" control flows that may persist after the use of the aforementioned primitives, we employ the hidden primitive, which focuses on transforming direct control flows into indirect ones. For example, we replace the branch statement with a throw statement at the end of basic block ② in `sepFunc-1`. Subsequently, catch statements are added in the following basic blocks (ⓓ and ③). By utilizing exceptions and the associated catch statements, we achieve an indirect control flow while preserving the intended functionality of the code. Furthermore, we extend our proposal to include equivalent hiding methods for the C language and call statements. These methods are discussed in detail in §3.4. Lastly, to further break the boundary of `hidFunc-1` with other functions, we divide the hidden code (ⓓ and ③) into different functions. This step ensures that the control flow is distributed across multiple functions, contributing to the overall obfuscation and making it more challenging to analyze the program structure.

Changing functions by recombining basic blocks from different functions is not trivial, and it still faces several challenges in performance, correctness, and obfuscation.

- **Challenge-1:** Choosing which basic blocks (or functions, control flows) to be obfuscated will seriously affect the performance overhead and obfuscation effect, and how to balance them well is difficult. For example, separating each basic block as a *sepFunc* would favor the obfuscation, but bring unacceptable overhead.
- **Challenge-2:** How to completely rebuild all control flow and data flow among functions after transformation (especially the fusion) is difficult. For example, once several functions participate in the fusion, we need to handle all pointers of the *oriFunc*s so that it can correctly jump to the *fusFunc* when de-referenced.
- **Challenge-3:** The hidden codes remain in the original function after the throw-catch process, which does not break the function boundary, it is challenging to escape them into other functions without changing the compiler architecture or influencing the program's original functionalities. Besides, though C provides an exception-handling mechanism via library functions (setjmp()/longjmp()), it brings patterns unavoidably, how to achieve a similar hiding effect as C++ while avoiding introducing new patterns is also challenging.

The following subsections detail the three primitives' design and address the above challenges.

## 3.2 The Fission Primitive

Fission focuses on the inside of every function and is mainly divided into 3 steps: (1)region identifying; (2)data flow and control flow rebuild; and (3)function-specific data-flow reduction.

*3.2.1 Partitioning Regions to Form sepFunc.* In general, a function's property is a single entry and multiple exits. Hence, as long as a certain code region satisfies this property, it can be separated to become a new function. More precisely, as long as a code region is a dominator tree in the control flow graph, it can be extracted into a *sepFunc*. The fission creates call relationships among *sepFunc*s and *remFunc* to ensure correctness. If the fission generates too many *sepFunc*s, the newly created function calls in *remFunc* will bring additional overhead (especially new function calls inside a loop). However, if the number or size of the *sepFunc*s is small, the *oriFunc* cannot be significantly changed. Therefore, designing a reasonable region identification algorithm is the key to reducing the overhead and improving the obfuscation effect.

**The region identifying algorithm**. We abstract the code region partitioning problem as a graph-cutting problem. The function's control flow graph can be regarded as a directed graph, and the edge weight represents the frequency of execution which indicates the cold/hot information. Partitioning the code region can be regarded as cutting the graph, where the weight of the cut edge is the cost of performance and the obfuscation effect is the number of the nodes in the sub-graph. Based on the above idea, we design the region-identifying algorithm on top of the directed weighted graph cut algorithm [57] to balance the performance overhead and the obfuscation effect. The algorithm takes function code as input and performs dominator tree analysis [36] at first. To avoid separating the whole function body into a *sepFunc*, we remove the dominator tree of the function itself and identify the regions from the rest of the trees. To indicate the effect of fission on obfuscation, we use the number of basic blocks in the tree to represent it. To indicate the effect of the fission on performance, we use the execution frequency of the root node of the dominator tree by using block frequency analysis [39] and the loop count (if the region is in a loop, the call to *sepFunc* will increase) as the cost of the cut. We iteratively select the most cost-effective (i.e., maximum ratio of effect and cost) dominator tree to separate until the tree set is empty.

*3.2.2 Data-flow Rebuild.* In addition to identifying regions as the function bodies of *sepFunc*s, we also need to identify the inputs and the outputs of these regions to construct the parameters and the return value of *sepFunc*s. For each variable used in a region, it should be input if its point is outside the region; Similarly, for each variable defined in a region, it should be output if it has a use point outside the region. For example, as shown in Fig. 2, the `fd` and `n` variables are inputs because the defined points are outside the region, and the `value` variable has a use point outside the region, so it is an output. For the variables whose define-use relationships are across

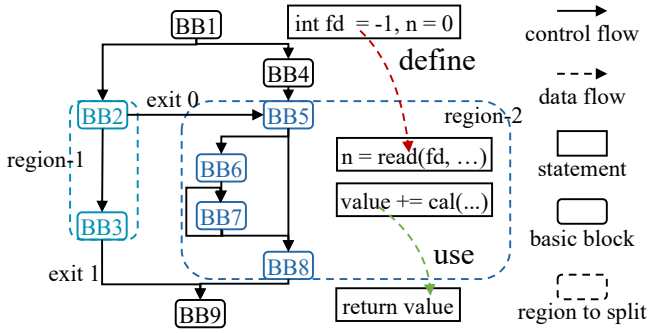


Fig. 2. The control-flow and data-flow graphs of cal_file() in Fig. 1

regions, we use the function parameters to pass the pointer to them. We don't pass a region's output variables by using the return value of *sepFunc* because a region may have multiple output variables.

**Data-flow reduction.** In general, the local variables of a function are defined at the entry basic block. Therefore, if an identified region needs to use local variables, these variables need to be passed into the *sepFunc* through parameters. In fact, if some local variables are only used by a *sepFunc*, then these variables do not need to be passed into the *sepFunc*, they can be defined directly in the *sepFunc*. This can shorten the length of the *sepFunc* parameter list, save unnecessary variable transmission, and further improve performance. To achieve this, we propose a lazy allocation strategy — if a local variable is only used in the region, we will move the variable definition to the *sepFunc*. For example, the `n` variable in Fig. 2 is initially defined in the *oriFunc* but redefined and only used in the `region-2`, which becomes `sepFunc-2` function, so the definition point of the variable can be delayed in the `sepFunc-2` function.

*3.2.3 Control-flow Rebuild.* We extract the basic blocks of each identified region into a *sepFunc*. The jump relationship between the regions in the *oriFunc* is transformed into the function call-return relationship after fission. The creation of a function call is simple, we only need to insert the function call at the location of the entry basic block of the region before extraction and set the parameters that need to be passed into the *sepFunc*. To further enhance the performance as well as the obfuscation effect, we additionally add the *ALWAYS_INLINE* attribute to the *remFunc*. Since the fission reduced the size of the *remFunc*, it can be inlined easily.

The handling of function returns is relatively complex due to: If a region has multiple exits, the corresponding *sepFunc* needs to encode this information into the return value, so that the *remFunc* can use this information to select the corresponding code to execute. As Fig. 2 shows, for the two exits (`0` and `1`) in `region-1`, when `sepFunc-1` returns from `exit 0`, the control flow should go to BB5, and when returns from `exit 1`, it should go to BB9. We use the return value of *sepFunc* to indicate the *remFunc* to determine the execution direction: We first number each exit of the *sepFunc*, use the number as its return value, and then insert a basic block at the call-site of this *sepFunc* in the *remFunc* (e.g., ⓐ in Fig. 1) to transfer control flow based on the return value.

*3.2.4 Handling the Exception Control-flows.* During program execution, there are some exception control flows that deviate from the usual function call and return, including the `setjmp/longjmp` and the C++ exception handling. The fission requires special handles of them.

**Handling the setjmp/longjmp.** Programmers could use the `setjmp()` to record the current context into a `jmpbuf` structure. And then, they could use the `longjmp()` in any subroutines on the call chain of this function to go back to the place the `jmpbuf` is pointing to. There is a requirement here that the `setjmp()` and the `longjmp()` using the same `jmpbuf` must be in the same call chain. Therefore, the call-site of the `setjmp()` cannot be separated into any *sepFunc*, because the stack frame of the function that calls the `setjmp()` cannot be freed when the corresponding `longjmp()` is executed. Otherwise, the `longjmp()` will direct control flow to an unknown location.

```
1  void bar(short a, float b) {
2      // bar's code
3      printf("bar: %d, %f\n", a, b);
4  }
5  int foo(int m) {
6      // foo's code
7      printf("foo: %d\n", m);
8      return m;
9  }
10 int main() {
11     bar(0x1234, 0.1);
12     int res = foo(1);
13     ...
14 }
```

(a) Before fusion

```
int bar_foo_fusion(int ctrl, short a, float b, int m) {
    if (ctrl) { // bar's code
        printf("bar: %d, %f\n", a, b);
        return 0;
    } else { // foo's code
        printf("foo: %d\n", m);
        return m; }
}
int main() {
    // ctrl is 1, executing bar
    bar_foo_fusion(1, 0x1234, 0.1, 0);
    // ctrl is 0, executing foo
    int res = bar_foo_fusion(0, 0, 0.0, 1);
}
```

(b) Fusion w/o parameter compression

```
int bar_foo_fusion(int ctrl, int x, float b) {
    if (ctrl) { // bar's code
        printf("bar: %d, %f\n", (short)x, b);
        return 0;
    } else { // foo's code
        printf("foo: %d\n", x);
        return x; }
}
int main() {
    // ctrl is 1, executing bar
    bar_foo_fusion(1, 0x1234, 0.1);
    // ctrl is 0, executing foo
    int res = bar_foo_fusion(0, 1, 0.0);
}
```

(c) Fusion w/ parameter compression

Fig. 3. An example of performing the fusion on two functions.

**Handling the C++ exception.** The exception-handling mechanism is a key feature of C++ that developers can capture actively or passively generated exceptions in the `try` block by writing the `catch` statements. It requires the cooperation of the system ABI and the exception-handling runtime library. They select the appropriate handling code to execute by searching the exception-handling information recorded in the binary. For the Itanium exception handling on Linux and the SEH mechanism on Windows, exception-handling information is recorded at function granularity. Since the fission moves part of the code into a *sepFunc*, the `try-catch` pair may be broken, making exception-handling information inconsistent. Simply skipping the exception-relevant code (avoiding them from being separated into any *sepFunc*) would reduce the obfuscation effect. Therefore, when identifying the code region, if it contains any code that may generate an exception, we will locate the corresponding `catch` code and divide it into the region at the same time.

## 3.3 The Fusion Primitive

The fusion selects functions to form *fusFunc* and rebuilds the control and the data flow to ensure correctness. While fission focuses on the inside of the function, fusion pays much more attention to the outside. This nature of fusion leads to the function-choosing problem, which contains two folds. One is how many functions to choose for fusion at a time, and the other is which functions to choose. In theory, the fusion can aggregate any number of functions. To balance the performance overhead and the obfuscation effect, we choose to aggregate two functions to form a *fusFunc*.

*3.3.1 Selecting Functions to Form fusFunc.* The fusion cannot arbitrarily select functions, it needs to select functions with compatible types of return values. The definition of incompatibility is that if converting between two types loses precision, the two types are incompatible. For example, when the return value of one function is an integer and the other is a float, these two functions cannot be aggregated. In fact, other conditions limit the selection of functions: 1) The variadic functions, e.g., the `printf(...)`; 2) Two functions with incompatible types of the return values; 3) Two functions that have direct calling relationship. The first two constraints are designed for correctness, and the last is designed for performance to avoid generating a lot of recursive *fusFunc*s. Functions that meet the above constraints will be randomly aggregated in pairs.

*3.3.2 Data-flow Rebuild.* Once the two functions to be aggregated are determined, the function prototype of the corresponding *fusFunc* can be determined immediately. For example, as shown in Fig. 3 (a) and (b), the `bar()` and the `foo()` are aggregated into `int bar_foo_fusion()`. The `ctrl` parameter is used to select the function bodies aggregated from the `bar()` and the `foo()`. Determining the function prototype of *fusFunc* is crucial to the rebuild of the data flow, which involves setting the parameter list and return value.

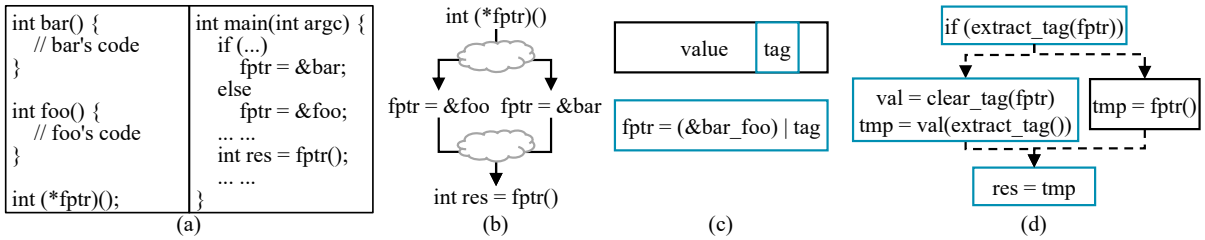


Fig. 4. Function reference and indirect call processing.

**Parameter list compression.** Simply grouping the parameter lists makes the parameter list of *fusFunc* too long, which will degrade the performance of calling *fusFunc* because too many parameters will be passed on the stack, which is an inefficient way. To achieve efficient parameter passing, we propose a parameter list compression mechanism — if the types of the two parameters from the two functions are compatible, we compress them into one. The reason why we can do this is that when a *fusFunc* is called, only the parameter list of one of the functions participating in the aggregation is used. For example, as Fig. 3(c) shows, both the `bar()` and the `foo()` have an integer parameter (`short a` and `int m`), we compress them into one integer parameter (`int x`). Parameters that can not participate in the compression will be copied into the parameter list of the *fusFunc*. In the worst case, the number of parameters is the sum of the parameters of the two functions, which means none of the parameters can be compressed. To avoid using the stack to pass parameters as much as possible, we preferentially select functions with a total number of parameters less than six for the fusion.

**Return value determination.** Determining the return type of *fusFunc* is relatively simple: 1) If the return type of one function is `void`, then the return type of the *fusFunc* is the return type of another; 2) If the return types of the two functions are both not `void`, the compressed type is used as the return type of the *fusFunc*, which is similar to the parameter list compression mechanism.

*3.3.3 Control-flow Rebuild.* Once the *fusFunc* is created, the two involved *oriFunc*s are removed, and all call-sites to the *oriFunc*s are replaced to call the *fusFunc*. As mentioned before, a `ctrl` parameter is added to the *fusFunc* to select the code block aggregated from the *oriFunc*s. The value of this parameter is `0` or `1`, which is set according to the original call site of the *oriFunc*. Since the *fusFunc* parameter list includes the parameters of both *oriFunc*s, we only need to pass the parameters required by the *oriFunc* to the *fusFunc* at the call-site of this *oriFunc*. Unused parameters are set to be `0`. Indirect function calls mainly include two cases: one is calling a function by dereferencing a function pointer, and the other is calling an exported function outside the module.

**Handling Indirect function calls.** Indirect function calls are more difficult to handle than direct function calls because we do not know where the *oriFunc* will be called. Fig. 4 (a) shows an example that calls two functions by de-referencing the function pointer. The corresponding data flow is given in Fig. 4 (b). When aggregating the `bar()` and the `foo()`, we need to change the function pointer points to the *fusFunc* and then replace the function call to call this *fusFunc*. However, we encounter a problem in that we do not know what the value of the `ctrl` parameter should be set to. This is because, at the compile time, we don't know whether the original function pointer `fptr` points to the `bar()` or the `foo()`.

To address the above problem, we propose a tagged pointer mechanism, which is similar to the low-fat pointer [35]. The core idea is to encode the information (called *tag*) of which *oriFunc* pointed to by the original function pointer into the new function pointer, and when the new function pointer is de-referenced to make a call, the value of the `ctrl` parameter can be dynamically determined by parsing the new function pointer. As shown in Fig. 4 (c), when the operation of taking the address of the function participating in the aggregation occurs, we need to perform the encoding operation. Since the *tag* is encoded into the function pointer, it can be propagated along with the function pointer. When the function pointer is de-referenced to make a call, we will extract the *tag* in the pointer and set the `ctrl` parameter according to the *tag*.

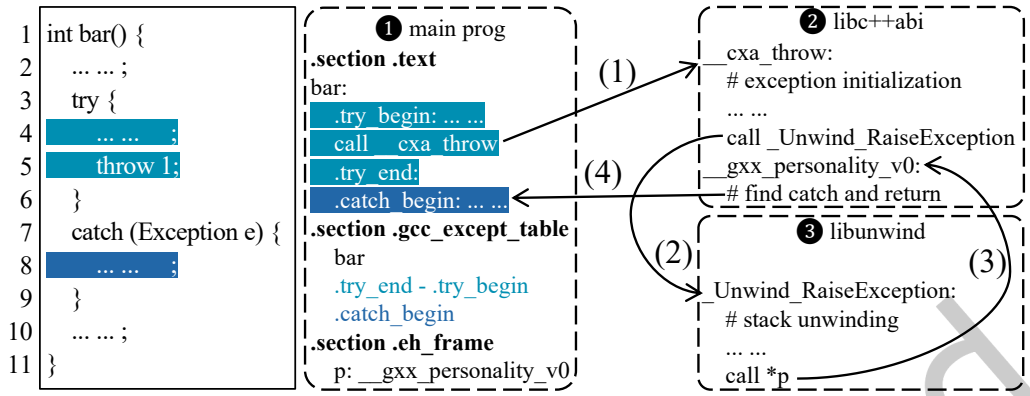


Fig. 5. The exception-handling mechanism for Linux C++.

The *tag* requires two extra bits, where a bit indicates whether the pointer points to a *fusFunc*, and the other bit records the value of the `ctrl` parameter. For example, if the pointer `fptr` points to the `bar()`, the *tag* will be set to 11b. As shown in Fig. 4 (d), when the pointer `fptr` is dereferenced to make a call, we insert code to check whether the *tag* is empty. If not, the code will extract the `ctrl` parameter and call the *fusFunc*. Otherwise, no additional operations are required.

## 3.4 The Hidden Primitive

The purpose of the hidden primitive is to transform direct control flow into *data mappings* and employ indirect control flow to counter the analysis. While fission and fusion aim to break function boundaries, hidden seeks to blur them. The hidden primitive leverages two indirect control flow methods, namely the *exception handling mechanism* and the *virtual call mechanism*, to hide the control flow graph (branch instructions) and the call graph (call instructions) respectively.

*3.4.1 The Control Flow Graph Hiding.* The hidden primitive utilizes the cross-module exception-handling mechanism to hide branches within functions. We first introduce the branch selection process for concealing control flow, then we discuss the details of the obfuscation method for C++ programs. For C programs that lack native exception handling, a complementary approach is used. To address inter-procedural considerations, a code dealer design ensures consistent and comprehensive control flow obfuscation. By combining these techniques, the hidden primitive offers robust protection against reverse engineering and code analysis for both C++ and C programs.

**Selecting Branches to Hide.** The selection of branches is decided in three folds: 1) hide the sub-graph of the CFG as large as possible; 2) minimize overhead caused by exception handling; and 3) preserve the program's functionality.

First, to hide the major part of the control flow, the general principle is to conceal the control flow as early as possible. This approach is based on the fact that *the code behind the obfuscated branch has no code reference in the control flow but only data references, eliminating the need for repeated concealment.* To this end, the first branch of every function should be hidden.

However, when dealing with functions with complex control flow structures, such as loops that include nested loops or branches, if its first branch is on the critical path, transforming it into the exception-based form would result in significant overhead. To address this issue, we chose to only obfuscate the *non-taken* branch of the loop entry and the *taken* branch of the loop exit, while leaving the branches inside the loop body untouched. Additionally, considering the loop may contain multiple exits, all the *taken* branches of exits are obfuscated. Identifying exit branches is accomplished by locating all the branches inside the loop with an outside destination.

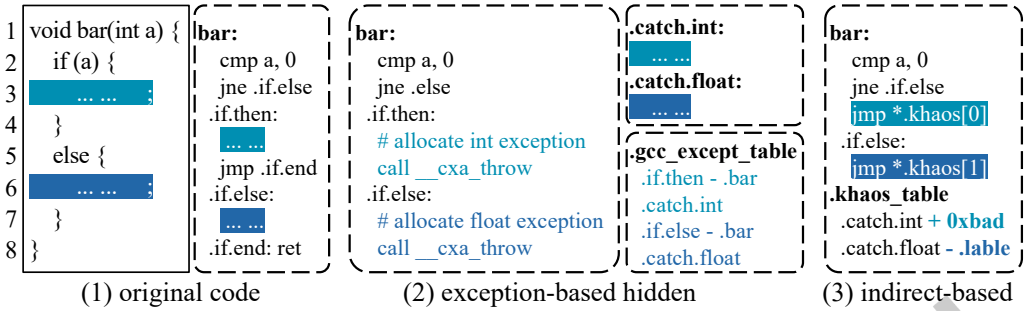


Fig. 6. The hidden example for branch instructions.

This trade-off is based on our observation of the loop's structure: *From a flow graph perspective, the loop can be visualized as a circle with multiple nodes, and this circle can be collapsed into a single node, simplifying the function into a sequential form.* Based on our experiments (§5.5), this design effectively hides the major part of the control-flow structures in most functions.

At last, to preserve the functionality, the branches that are already exception-related (e.g., branches in try blocks) are skipped so as not to interfere with the original exception relationship.

**The C++ exception-handling in Detail.** As mentioned in §3.2.4, the exception-handling mechanism relies on the system ABI and the exception-handling runtime library. The Fig. 5 depicts the simplified C++ exception mechanism under the Linux ABI, which requires the compiler to generate exception-relevant information in the binary (❶ main prog), the C++ language-specific exception-handling library (❷ libc++abi), and a language-independent but OS-dependent stack unwind library (❸ libunwind).

As shown in Fig. 5, for the throw statement that actively generates an exception, the compiler compiles it into a direct call to the __cxa_throw() function in the libc++abi library. This call contains three parameters, which are the exception object, exception object type information, and the pointer to the exception object's destructor. Then the __cxa_throw() function initializes the exception object and calls the _Unwind_RaiseException() function in libunwind, which retrieves the address of the code that throws the exception, and unwinds the stack frame through the **eh_frame** section in the binary. During the unwinding process, _Unwind_RaiseException() function indirectly calls the __gxx_personality_v0() function (C++ related) defined in libc++abi. __gxx_personality_v0() searches for the catch statement under the current stack frame by querying the **gcc_except_table** in the binary. Finally, it replaces the return address on the stack with the address of the corresponding catch statement, effectively redirecting the control flow to the appropriate catch block.

**The Hiding Process.** The hidden primitive utilizes the above cross-module mechanism to hide the control flow. First, it numbers the basic block inside the function. The number indicates the type of exception thrown after the basic block is executed, such as integer, floating point, and customized types. Then, it inserts the corresponding catch statements in the subsequent basic blocks to distinguish different control flows according to the type.

For example, in Fig. 6 (1) the original control flow relationship between different basic blocks is apparent. After the control flow is hidden (Fig. 6 (2)), the original basic blocks are replaced with throw statements. During program execution, the control flow becomes intertwined with the system ABI and the exception-handling runtime library. These components work together to select the appropriate handling code by searching the exception handling information in the binary file. Other obfuscation techniques like data obfuscation can be used here to enhance the hidden effect, for example, by adding or subtracting a random number to the `.gcc_except_table`'s item. Since this paper focuses on code transformation, we leave this as an orthogonal technique.

**The Complementary Method for C Programs.** The exception-handling mechanism is primarily designed for C++ programs, but it is important to recognize the need for a similar hiding effect in C programs. This becomes particularly crucial when considering the extensive codebase of C programs, their vital role in modern

computer systems, and their popularity in various utilities. Additionally, the research of static code analysis and vulnerability detection also heavily focused on C programs [4, 7, 37], especially when studying on large code bases [51].

To extend the coverage to C programs, we analyze the essence of the exception-handling mechanism and abstract it as a *table-driven lookup-based indirect jump* statement. In summary, the mechanism searches for exception-handling information generated by the compiler, identifies the address of the corresponding catch statement, and performs a jump to that location.

Building upon this observation, we propose a complementary method specifically tailored for C programs. As Fig. 6 (3) shows, firstly, we create a reference table called `.khaos_table`, which simulates exception-handling information and generates a corresponding set of jump targets. Next, the code is reinforced through indirect jumps. Simultaneously, the simulated set of jump targets is relocated at load time similar to the exception-based hidden method.

Upon applying the hidden primitive to the code, numerous code fragments are not referenced by any other code but by data. These fragments constitute isolated and unreferenced portions of the program, further enhancing the concealment of the control flow. By employing this complementary method, we extend the benefits of the exception-based hidden approach to C programs. The construction of the `.khaos_table` and the utilization of indirect jumps contribute to the creation of isolated code fragments, making the control flow more obscure and difficult to analyze.

**The Hidden Code Dealer.** After the indirect control flow is generated, the hidden code remains in the original function. To fulfill the inter-procedural obfuscation, we intend to send these codes to other functions. However, it is challenging to do it without influencing the program's functionality in the current compiler architecture. Take the LLVM framework as an example, inter-function transformations are typically performed in the middle end using module passes. These passes operate on all functions within the same compile unit (e.g., .cpp file). However, at this stage, the program code is in a Static Single Assignment (SSA) form, and moving code from one function to another can disrupt the SSA form, resulting in inconsistent program functionality.

Moving to the LLVM backend, where the code is in a non-SSA form, might seem like an option. However, the backend processes only one function at a time. It transforms an SSA function into the non-SSA form, emits its binary code after some intra-function transformations, frees the non-SSA function, and then moves on to the next SSA function. This means that there is only one non-SSA function available at any given time, making direct code manipulation challenging.

To overcome these challenges, the copy-delete-based code dealer approach is proposed. This approach aims to move the hidden code from its original function to another function while preserving the program's functionality and adhering to the constraints of the compiler architecture. It involves copying the hidden code to another function during the middle end, where inter-procedural transformations are possible. It additionally adds references to the copied code to prevent it from being optimized away. Once the non-SSA form code is generated, the original code in the *hidFunc* is deleted, and all references to the original code are adjusted to point to the corresponding copy. Importantly, the reference adjustment is performed intra-procedurally, only requiring changes to the referenced symbol within the *hidFunc*.

By employing this approach, the hidden code can be correctly moved into other functions without altering the underlying compiler architecture. As a result, the program will contain numerous indirect inter-procedural control flows that jump to the middle of functions in an interleaved manner, further obfuscating the control flow and making the program more challenging to analyze.

*3.4.2 The Call Graph Hiding.* The above hiding process has covered the jump relationship inside the functions but leaves the call relationships untouched. Similar to the exception handling mechanism, which decides the control flow at runtime, there is a dynamic dispatching mechanism for function calls in C++ — *virtual tables* (vtable)

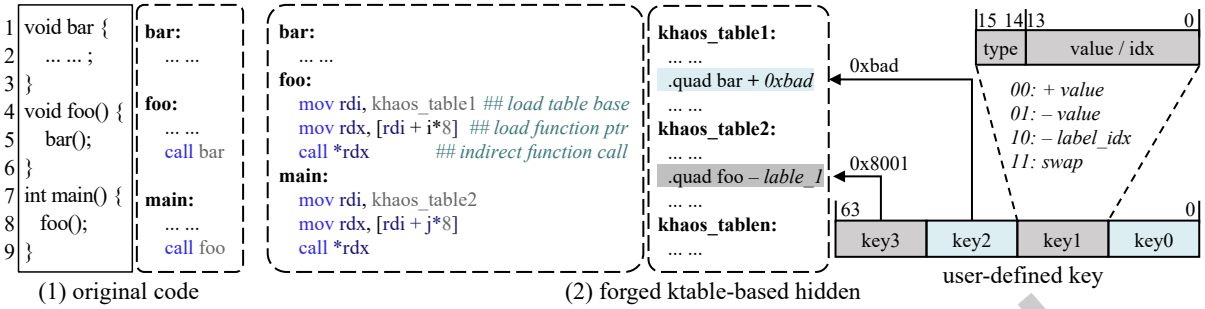


Fig. 7. The hidden example for call instructions.

mechanism, which is used for runtime polymorphism. Inspired by the *counterfeit object-oriented programming* attack [55], which induces malicious program behavior by forging illegal vtable that contains legal virtual functions pointers, we propose the call graph hiding by forging vtables and transforming the calls into virtual function call form.

**Selecting Calls to Hide.** Selecting calls to hide poses a different challenge compared to branch selection. Unlike branches that are contained within a single function, calls can be scattered across different functions, making it necessary to consider all instances of the call to achieve a comprehensive hiding effect. Hiding only the first call instruction would leave other call instructions visible and potentially expose the underlying control flow.

To achieve a comprehensive hiding effect within the call graph, it is imperative to obfuscate as many calls as feasible. Nonetheless, selecting calls entails a delicate balance between performance optimization and obfuscation. Therefore, we have embraced a design principle akin to the fission primitive: by identifying critical code segments or sensitive functions where heightened obfuscation is essential for security, we intensify obfuscation techniques in these areas. Simultaneously, we exercise caution in less critical sections to mitigate performance impact.

This methodology ensures that calls with minimal performance implications are prioritized for obfuscation, thereby striking an equilibrium between effective concealment and maintaining reasonable program execution efficiency. Additionally, alternative strategies such as utilizing profiling tools to pinpoint performance bottlenecks introduced by obfuscation, fine-tuning obfuscation parameters for optimized code, and progressively integrating obfuscation into the codebase while monitoring performance impacts at each stage, can be employed to manage this delicate tradeoff, we leave them as future work.

**The Hiding Process.** The hidden primitive randomly fills function pointers corresponding to callee functions into forged tables. At the call site, the original call instruction is transformed into the virtual function call form by replacing the call target regarding the vtable. The modified call instruction specifies the target vtable and the index of the desired function within that vtable. For example, in Fig. 7 the original code contains three functions with direct call relationships. Firstly, the hidden primitive fills the function pointers to *bar()* and *foo()* into the item *i* of *khaos_table1* and item *j* of *khaos_table2*, respectively. Secondly, the original direct call instructions are transformed as function pointer loading instructions and indirect calls with the corresponding vtable and index.

**Arrangement of Ktables.** The arrangement of forged ktables considers both the calls within the same basic block and those in different functions, aiming to optimize performance and enhance obfuscation. For calls within the same basic block, grouping the targets within the same ktable improves localities. This means the calls made within a specific code section are organized together, allowing for better data cache utilization and potentially reducing the overhead associated with repeated lookups. On the other hand, calls in different functions, even if they have the same call target, are placed in separate tables. This approach ensures that the ktables remain function-specific and prevents potential analysis from identifying patterns or relationships between functions easily.

**Obfuscation for Ktables.** Similar to the control flow graph hiding approach, the elements within vtables are obfuscated during compilation. Specifically, the user specifies a 64-bit key, which contains 4 16-bit sub-keys, and the number of every forged vtable is multiple of four. Each sub-key includes 4 cases, indicated by its most significant two bits as follows:

- **00/01:** add/subtract a constant to the function pointer, the constant value is decided by the lower 14 bits in the key;
- **10:** subtract a label defined by KHAOS, the label number is decided by the lower 14 bits;
- **11:** swap the current item with the next item in the same vtable.

KHAOS inserts an initialization code into the program with the user-defined key. It parses the key at the program loading time to sequentially recover the actual pointers in the forged vtable. This ensures that the actual function pointers are concealed. The obfuscated elements are corrected at the program loading time, allowing the program to function correctly. This way, the control flow is obscured, and the actual function being called is determined dynamically at runtime.

### 3.5 Combination

The three primitives, fission, fusion, and hidden, possess distinct obfuscation characteristics and can be combined to enhance the obfuscation effect. Each primitive serves a specific purpose in altering the program's structure and control flow.

- Fission excels at dividing large functions into smaller ones, but it may be less effective when applied to small functions. Its primary goal is to split functions and introduce trampoline blocks to establish call relationships between the separated parts.
- Fusion, on the other hand, is suitable for combining two functions of similar size to obscure their individual features. When applied to functions of disparate sizes, fusion may not significantly alter the characteristics of the larger function. The main objective of fusion is to aggregate code blocks and adjust references to create a consolidated function.
- Hidden builds upon the foundations of fission and fusion. Its primary aim is to conceal the transformations performed by fission and fusion at compile-time and reveal them at runtime. Hidden focuses on transforming direct control flow relationships into exception-handling mappings and employs indirect control flow to impede analysis tools.

By combining the three primitives, developers can achieve a comprehensive and robust obfuscation effect. Fission and fusion primarily target the program's control flow graph, while hidden adds compile-time hiding and runtime recovery mechanisms for deeper obfuscation. We propose an *auto mode* combination. which first attempts to split functions as much as possible using fission, then utilizes fusion to aggregate the *oriFunc*s that were not processed by fission, and applies hidden to cover the dominator paths in the program. The auto mode combination strikes a balanced trade-off between obfuscation effectiveness and performance overhead. It is well-suited for real-world software scenarios. Developers can choose this combination or select the appropriate combination based on their specific requirements. For example, by only aggregating and hiding the *sepFunc*s generated by the fission process, the issue of handling indirect function calls is eliminated, which is particularly suitable for programs with numerous exported functions, such as a runtime library.

## 4 KHAOS Implementation

The KHAOS is implemented based on the LLVM-9.0. Fission, fusion, and hidden are middle-end passes, with the code dealer containing a backend pass, and the fusion pass is scheduled after the fission pass and before the hidden pass. KHAOS takes the source code and the user's parameters as input, which specifies the obfuscation

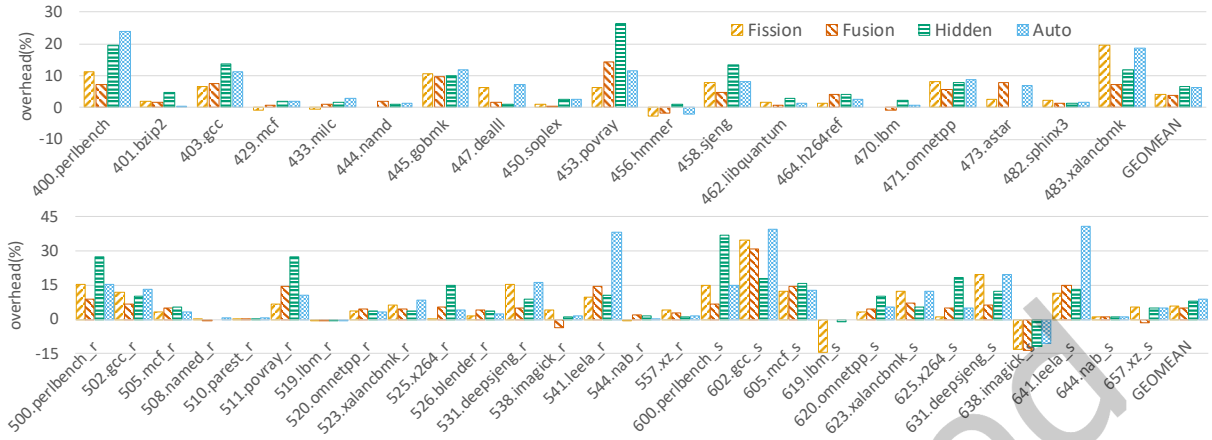


Fig. 8. Runtime overhead of SPEC CPU 2006 (upper part) and 2017 (lower part) C/C++ programs.

options (e.g., the selection of the primitives). The source code will be compiled into the LLVM IR, and then KHAOS performs the transformations and compiles them into the executable.

**The tag bits choice.** As mentioned in §3.3, the tagged pointer selects the code block aggregated from different *oriFunc*s. On the X86_64 architecture, only 48 bits of the virtual address are effective, so the upper 16 bits of the function pointer are unused and can be used to place the *tag*. However, this approach is expensive when handling statically initialized pointers, such as global static function pointers and virtual function tables. For the position-independent executable, these pointers need to be relocated at load time. To attach the *tag* information to them, we need to add an initialization code to rewrite these pointers after the relocation, which will slow down the loading process.

To address this problem, we choose the lowest bits of function pointers. This is because the functions are usually 16 bytes aligned with the performance consideration, so the lowest 4 bits of the function pointer can be used to place the *tag*. Actually, the clang compiler has already used the least bit to identify whether a function pointer points to a virtual function or not, so currently, only the 3 bits are unused. Instead of rewriting statically initialized pointers after the relocation, we utilize the relocation mechanism directly by adding the *tag*'s value to the addend field (which is used to add an offset when relocating) of the relocation item, so the *tag* can be attached to the pointer during the relocation. This method cannot be applied to support the upper bits *tag* because it exceeds the range supported by the addend field, i.e., $(-2^{31}, +2^{31}]$.

## 5 Evaluation

We run KHAOS on Ubuntu 22.04 (Kernel v5.15.0) that runs on an Intel(R) Xeon(R) Gold 6148 CPU with 160 cores and 1.5TB memory. This section evaluates KHAOS in terms of effectiveness and performance, and answers the following questions:

- (Q1) How is the performance of the obfuscated programs?
- (Q2) How does KHAOS work against the state-of-the-art binary diffing techniques?
- (Q3) How good is KHAOS at hiding real vulnerable code?
- (Q4) How good is KHAOS compared with other obfuscators?

**Test Suites.** We used three test suites to evaluate KHAOS: 1) All C/C++ programs in SPEC CPU 2006/2017 benchmarks with the *ref* input (denoted as the **T-I**); 2) All 108 programs in the CoreUtils 8.32 (denoted as the **T-II**); 3) Five commonly used programs in embedded devices with at least one vulnerability, including two popular

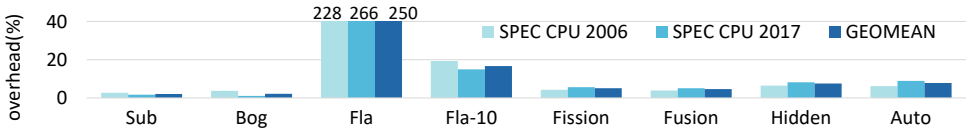


Fig. 9. Runtime overhead of O-LLVM and KHAOS.

IoT JavaScript engines (JerryScript and QuickJS), OpenSSL-1.1.1, BusyBox-1.33.1 and libcurl-7.34.0 (denoted as the **T-III**). The performance evaluation was performed on the **T-I** (Q1); The effectiveness against binary diffing techniques was evaluated on the **T-I** and the **T-II** (Q2); The ability to hide vulnerable code was evaluated on the **T-III** (Q3). Since software developers typically link programs into a single binary in embedded devices, we compiled and obfuscated these test suites in the same way under `O2` with the link-time optimization.

**Comparison targets.** To compare with existing obfuscator (Q4), we choose two popular compiler-level obfuscation tools (O-LLVM [32] and BinTuner [52], which are open-sourced and compiler-based (same as KHAOS)). O-LLVM [32] contains three obfuscation methods: instruction substitution (*Sub*), bogus control flow (*Bog*), and control flow flattening (*Fla*). Literature [3, 16, 52, 61] in software engineering, systems security, and programming languages fields all use it in their experiments. To ensure the consistency of the evaluation environment, we upgraded the LLVM version of O-LLVM [32] to 9.0, which is the same as KHAOS. We also choose BinTuner [52], which is an iterative compiler tool that uses compiler options to transform the code to enlarge the difference of binaries, as another target to compare KHAOS with the compiler's options.

## 5.1 Performance Overhead after Obfuscation

We separately evaluated the performance overhead of the fission, fusion, hidden, and the auto combination mode introduced in Section §3.5 on the **T-I**. As shown in Fig. 8, the geometric performance overhead of the three primitives and the auto mode are 5.1%, 4.6%, 7.5%, and 7.8%, respectively. The reason why some cases (e.g., `456.hmmer`) have a negative performance overhead is that after the fission separates part of the code, the *remFunc* can be further inlined to its callers, and the fusion can improve the code locality of the aggregated functions. The results demonstrated that obfuscations compliant with the compiler optimizations can have good performance advantages. We also compared the performance overhead of KHAOS with O-LLVM's *Sub*, *Bog*, *Fla*. As shown in Fig. 9, KHAOS has comparable overhead with the *Sub* and the *Bog*. Due to the high overhead of *Fla*, we reduce its obfuscation ratio to 10% (*Fla-10*), and others are all at 100%.

## 5.2 The Effectiveness against Binary Diffing

Table 1. Summary of the chosen binary diffing works and tools.

| **Works/Tools** | Asm2Vec[16] | DeepBinDiff[17] | DiEmph[65] | jTrans[59] | Optango[23] | Safe[43] | Trex[49] | Zeek[56] | BinDiff[73] | Catalog1[63] | FSS[18] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Approach**[1] | ✾ | ✾ | ✾ | ✾ | ✾ | ✾ | ✾ | ✾ | ✻ | ✻ | ✻ |
| **Granularity**[2] | F | BB/CG | F | F | F/CG | F | F/CG | BB | All | F | I/F/CG |
| **Symbol Relying**[3] | ✘ | ✔ | ✘ | ✔ | ✘ | ✘ | ✘ | ✘ | ✔ | ✘ | ✘ |
| **Mem Consuming**[4] | ✘ | ✔ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ |
| **Time Consuming**[5] | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ | ✘ | ✘ |

[1] The category of the corresponding work/tool. ✾: learning-based approach, ✻: traditional approach.
[2] Granularity for diffing. I: instruction, BB: basic block, F: function, CG: call graph, All: every granularity is considered.
[3] Whether the un-stripped binaries have side effects. [4/5] Whether the approach is memory/time consuming.

**Confrontation targets.** We leverage 11 state-of-the-art binary diffing techniques to evaluate the effectiveness of KHAOS. Their characteristics are summarized in Table 1. Among them, 8 approaches are learning-based and have been retrained on our test suites, while the remaining 3 are traditional approaches. For example, Catalog1 [63] and FSS (FunctionSimSearch [18]) are hash-based methods. Catalog1 uses raw bytes as input features and a

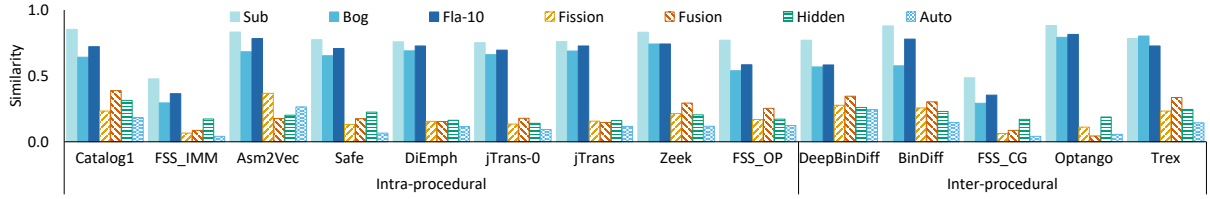


Fig. 10. Similarity results of chosen binary diffing works and tools.

different signature size (i.e., number of hash functions). FSS uses a combination of graphlet, mnemonic, and immediate. We followed the experiment setting in previous work [42] to evaluate them.

Although these techniques commonly employ function-based diffing granularity, they also have considered various other granularities during the diffing process. For example, the **DeepBinDiff** [17] chose the basic block as the granularity and embedded call graph information into its embedding. **BinDiff** [73] is an industry-standard binary diffing tool, which diffs the semantic similarity across multiple granularities, including instruction, basic block, function, and call graph). Moreover, **Optango** [23], **FSS** [18] (configured to use the graph in binary), and **Trex** [49] incorporate inter-procedural information including call relationship, call graph, and calling convention. Given that KHAOS focuses on inter-procedural obfuscation, the diverse range of the diffing granularities employed ensures a comprehensive understanding of strengths and limitations within the appropriate domain. *Symbol relying* means whether the un-stripped binaries have side-effects or not, for example, BinDiff usually uses function names to reduce the searching space. *Mem/Time consuming* means the tool requires a lot of memory (e.g., more than 1 TB) or takes a long time for retraining and diffing.

Comparing binary diffing works is challenging because their measurements of similarity are very different [52], such as graph edit distance or statistical significance. Simply comparing their similarity scores does not provide accurate information. For the open-sourced works in academia, we normalized their results by computing the ratio of true matching function pairs that are also the top-ranked matching candidates (i.e. *Precision@1*). IMF-SIM [61], Asm2Vec [16], and Ren et al. [52] also use Precision@1 to measure diffing accuracy in their paper. For the commercial binary diffing tool BinDiff [73], we normalized its similarity score to `[0, 1]`.

**Paring success judgment method.** Since KHAOS changes the number of functions, we relax the requirements for *Precision@1*. For the fission, if the *oriFunc* is paired with any *sepFunc*s generated from it or the *remFunc*, this pairing is recognized as successful. For the fusion, if the *fusFunc* is paired with any function before the fusion, this pairing is recognized as successful. For the fusion, if any part of the function is paired with the original function, this pairing is recognized as successful. For the tools that do not use the function as diffing granularity (e.g., basic block), the pairing is recognized as successful as long as their belonging functions are matched, even if the two basic blocks are not truly matched. It is worth noting that the above setting is looser than originally used in these tools but is more challenging for KHAOS.

**Test suite adjustment.** The test suites for DeepBinDiff [17] need to be adjusted due to the inability to run results. It requires too much memory (sometimes more than 10 TB) due to its representation of basic blocks. Since its diffing process is tightly coupled with binary size, we decided not to modify it and only use programs with less than 40k lines. Even with the reduced test suite, it is still time-consuming (e.g., over 1 week to diff binaries of `508.namd_r`). It's worth mentioning that this setting is unfavorable to KHAOS because it uses original functions to obfuscate each other, lacking material reduces the obfuscation effect. Other binary diffing tools still use the normal test suites.

**Results.** We evaluated the accuracy of these tools by comparing obfuscated and un-obfuscated (un-stripped) binaries on the **T-I** and **T-II**. As depicted in Fig. 10, higher accuracy means lower adversarial effect. Based on the corresponding approach, we grouped the results into intra-procedural and inter-procedural categories. On the one hand, for the binary diffing works that *only consider the intra-procedural information*, the inter-procedural

obfuscation can fundamentally defeat them because the code structures along with the semantics are significantly changed. On the other hand, for the binary diffing works that *take inter-procedure information into account*, the inter-procedural obfuscation can also defeat them because the inter-procedural information they extracted, such as the types of function calls, the numbers of function calls, and the call graph, are also significantly changed after the obfuscation.

Since DeepBinDiff [73] used the reduced test suite, which limits the ability of KHAOS, its result is higher than others. Although it uses the basic block level instead of the function level as its granularity, the feature vector of the basic block still encodes the control flow graph and call graph, which KHAOS has changed, and that's why KHAOS can defeat it. Since BinDiff [73] takes advantage of function names, its result is higher than others. With comparable overhead, KHAOS can achieve a much better adversarial effect than O-LLVM [32]. Importantly, KHAOS achieves this without introducing program-irrelevant code. This result indicates that KHAOS effectively enhances the obfuscation of the programs while maintaining their core functionality and relevance.

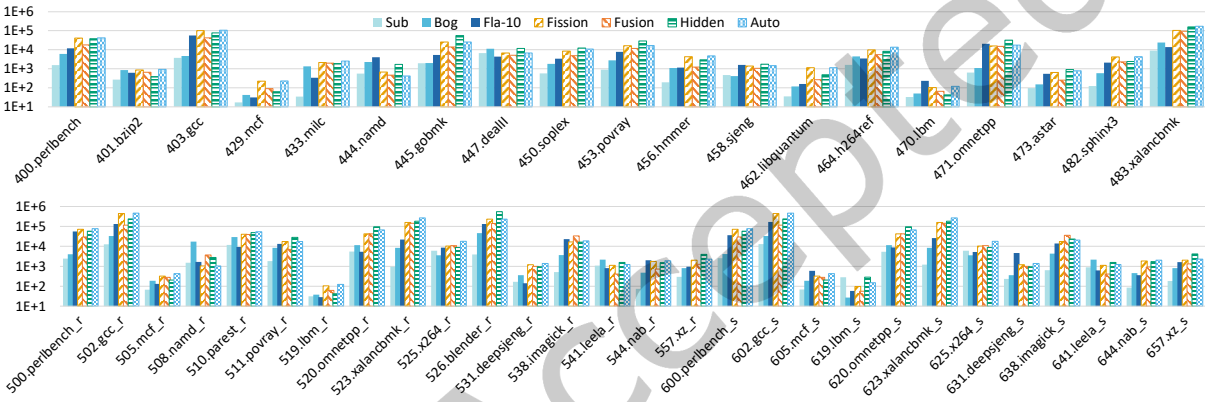


Fig. 11. The opcode vector distance between the original and obfuscated binaries.

**Opcode Histogram.** The opcode histogram is crucial internal information that can reveal the semantics of the entire binary. It has been extensively studied and discussed in various related works [6, 11, 12, 24, 25]. To this end, we present the opcode details collected from the **T-I** and **T-II** to further demonstrate the effectiveness of KHAOS. To obtain the opcode details, we utilized the *objdump* tool to disassemble all the binaries and collected the opcode histograms in vectors. Each element in the vector represents a specific opcode type (e.g., `add`), while its value indicates the frequency of occurrence for the corresponding opcode within the binary. By comparing the opcode vectors of the original and obfuscated binaries, we calculated the *vector distance*.

Fig. 11 depicts the calculated distances in a logarithmic scale, as different programs may have varying scales of opcodes. Notably, KHAOS can generate a wider range of opcodes for most programs. For example, in the case of `400.perlbench`, the distance of the binary generated by the *Auto* mode is 42,048, while O-LLVM achieved a maximum distance of 11,814. On average, the vector distance of KHAOS obfuscated binaries is 3.83 times greater than that of O-LLVM obfuscated binaries. This stark difference demonstrates the capability of KHAOS to diversify the opcode distribution effectively.

## 5.3 The Ability to hide Vulnerable Code

We use the **T-III** to evaluate the ability to hide 19 real-world vulnerabilities. In this experiment, we used Asm2Vec [16] and SAFE [43] to calculate the *escape@n ratio* (the rank of truly matched pair in the matched result) of vulnerable functions. The reason why other tools were not used is that they only give top-1 matched results. We calculated *escape@1/10/50 ratio* of vulnerable functions. For example, as shown in Fig. 12, the *escape@50 ratio*

of *Auto* on Asm2Vec is over 0.8, which means more than 80% of vulnerable functions can not be found within top-50 ranked functions. Moreover, the obfuscation ratio of *Fla* in O-LLVM is set to 100%, which would bring unacceptable overhead.

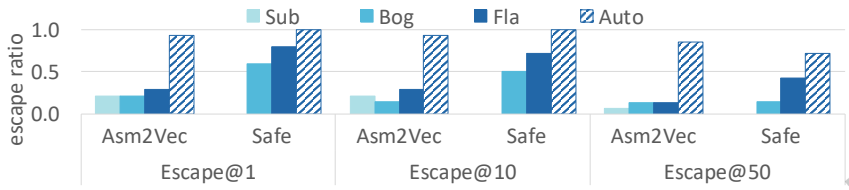


Fig. 12. Escape ratio for top@1/10/50 of vulnerable functions. Higher means stronger hiding ability.

The *escape ratio* could reflect the ability to hide the vulnerable code with different obfuscation methods. With the same precision and binary diffing tool (e.g., `escape@50-Asm2Vec`), *auto* is better than *Sub*, *Bog*, and *Fla* in O-LLVM. This ratio could also reflect the diffing ability of binary diffing tools. With the same precision and the settings of obfuscators, e.g., `escape@1-Auto`, Asm2Vec is more accurate than Safe. The experimental results show that KHAOS can not only fight against binary diffing tools but also reduce the pairing ranking of vulnerable functions significantly, achieving the purpose of hiding vulnerable code.

## 5.4 Comparison with BinTuner.

**Baseline setting.** The comparison with BinTuner [52] is a challenge because KHAOS and BinTuner cannot choose the same baseline. Suppose BinTuner chooses the same baseline as KHAOS (O2), to expand the binary difference, the generated binary would be close to O0, and the performance overhead would be higher, which is unfair to BinTuner. As a result, we followed the original baseline of BinTuner (O0) and used binaries generated by the *auto* mode for KHAOS.

**Binary diffing comparison.** We leverage BinTuner to generate the obfuscated binaries of **T-I** and calculate the similarity score of BinDiff [73] for the generated binaries by BinTuner and KHAOS. As shown in Fig. 13, the difference score for *BTR-O0* is the lowest among the BinTuner-generated binaries. This is because BinTuner uses O0 as the baseline during its iterative compilation process. On the other hand, *BTR-O1*, *BTR-O2*, and *BTR-O3* have higher difference scores. In the case of KHAOS, it chose O2 as its baseline, resulting in *Auto-O2* having a slightly higher difference score compared to *Auto-O0*, *Auto-O1*, and *Auto-O3*. However, all the KHAOS-generated binaries have much lower difference scores compared to the BinTuner-generated binaries.

It is worth mentioning that during the experiment, we observed a swinging phenomenon with BinTuner, where the generated binary, when using O0 as the baseline, was similar to O3, and when using O3 as the baseline, it was similar to O0. This suggests that BinTuner struggles to generate binaries with low overhead but significant binary differences. In contrast, KHAOS overcomes this limitation by leveraging the program itself to obfuscate it. Additionally, the fusion process in KHAOS is random, resulting in a different binary being generated each time. Overall, the binary diffing comparison provides strong evidence that KHAOS achieves a higher level of obfuscation with lower difference scores compared to BinTuner.

**Performance overhead comparison.** To compare the performance overhead of BinTuner and KHAOS, we collect their runtime performance compared with default optimization levels (O0 - O3). Both BinTuner and KHAOS have performance speed up on O0 and O1 while slowing down at O2 and O3. For the convenience of comparison, all the performances are normalized to [0, 1]. As shown in Fig. 14, BinTuner has comparable overhead with KHAOS in general, while it also brings significant overhead on some programs.

**Observation on compiler optimization regarding obfuscation.** Regarding the relationship between compiler optimization and obfuscation, compilers aim to improve program performance and reduce binary size through optimizations. However, code obfuscation techniques often need to counteract these optimizations to

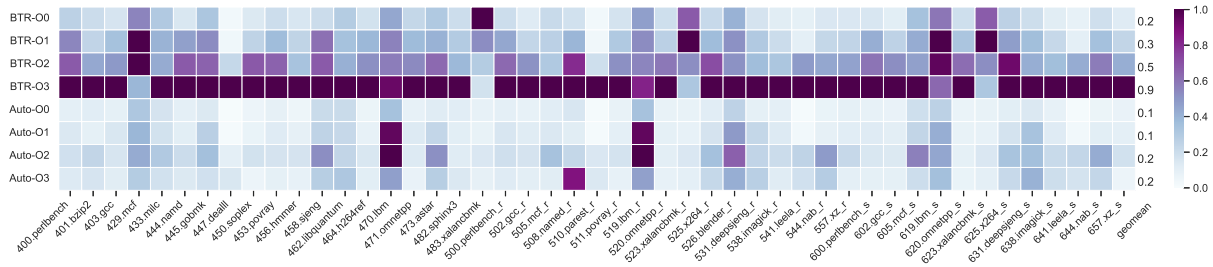


Fig. 13. BinDiff similarity score (normalized) of SPEC 2006 and 2017 C/C++ programs. *BTR-O0* corresponds to the similarity of binaries generated by BinTuner and LLVM O0. Lower is better.

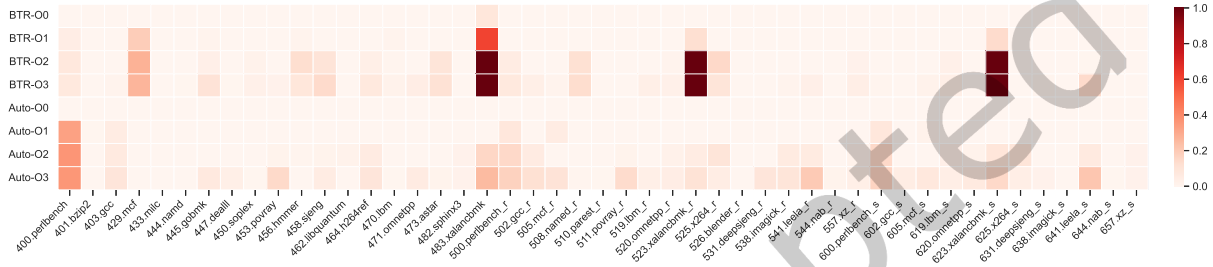


Fig. 14. Runtime overhead (normalized) of SPEC 2006 and 2017 C/C++ programs generated by BinTuner.

Table 2. Statistics of the hidden primitive.

| benchmark | program information | | | | control flow graph hiding | | | call graph hiding | | unhidden_ |
|---|---|---|---|---|---|---|---|---|---|---|
| | #func | #call | #loop | #call_in_loop | #hid_br | hid_ratio | CFG_coverage | #hid_call | hid_ratio | func_ratio |
| **SPEC CPU 2006 C** | 8,078 | 73,196 | 12,283 | 13,856 | 6,299 | 77.98% | 86.61% | 52,573 | 71.82% | 14.63% |
| **SPEC CPU 2006 C++** | 12,037 | 25,156 | 12,304 | 4,434 | 9,095 | 75.56% | 89.42% | 18,306 | 72.77% | 17.34% |
| **SPEC CPU 2017 C** | 27,174 | 356,696 | 48,644 | 84,355 | 22,417 | 82.49% | 87.90% | 251,057 | 70.38% | 16.76% |
| **SPEC CPU 2017 C++** | 63,009 | 150,146 | 52,848 | 27,869 | 39,590 | 62.83% | 87.78% | 99,784 | 66.46% | 19.25% |
| **Geomean** | - | - | - | - | - | 74.34% | 87.92% | - | 70.32% | 16.91% |



prevent the generation of identical binary code. This adversarial relationship between code obfuscation and compiler optimizations can lead to unacceptable performance overhead, as seen in the case of *Fla* in Fig. 9.

To minimize the overhead caused by obfuscation, the code obfuscation should comply with the compiler optimizations as much as possible. According to our statistics, over 85% of optimizations under LLVM's -O3 option are intra-procedural. Therefore, KHAOS, which leverages inter-procedural code obfuscation, performs better in terms of performance due to less interference with compiler optimizations. Furthermore, while compiler optimizations can transform the program, they are limited to specific patterns such as inline rules and loop-related transformation rules. This limits the flexibility of obfuscation. In contrast, KHAOS can transform the program in a much more flexible manner, providing greater versatility in obfuscation techniques.

## 5.5 The Statistics of KHAOS Internals

We collected the statistics of fission, fusion, and hidden individually without the combination. For **fission**, we calculated the fission ratio (#*sepFuncs* / #*oriFuncs*), which resulted in a ratio of 145%. This indicates more *sepFuncs* were generated compared to the *oriFuncs*. Additionally, we found that the average number of basic blocks in *sepFuncs* was 6.46. Furthermore, the reduced ratio of *oriFuncs* after fission was 42%, implying that a significant reduction in the number of *oriFuncs* was achieved.

For **fusion**, we measured the fusion ratio, which represents the ratio of successfully aggregated functions, resulting in a ratio of 97%. This high fusion ratio indicates that almost all functions were successfully aggregated during fusion. Additionally, we observed a reduced parameter number of 1.43 through parameter list compression. Furthermore, each function had an average of 1.89 innocuous basic blocks. It proves that both optimizations for runtime overhead (e.g., data-flow reduction) and obfuscation enhancement (e.g., innocuous analysis) have worked effectively.

For **hidden**, we measured the detailed obfuscation information in Table 2. The *program information* column provides the statistics of the original program, including the number of functions (*#func*), calls (*#call*), loops (*#loop*), and calls inside loops (*#call_in_loop*). We separately calculated the obfuscation results of the *control flow graph hiding* and the *call graph hiding* methods.

For the *control flow graph hiding* method, we calculated the hidden branch count (*#hid_br*) and the corresponding branch hidden ratio (*#hid_br/#func*). It is worth noting that C++ programs exhibit a lower hidden ratio than C programs, as the exception-related code is skipped. As explained in Section §3.4.1, our approach focuses on obfuscating branches as early as possible to conceal the major part of the control flow graph. To quantify the effectiveness of this method, we computed the coverage of hidden control flow graphs within the obfuscated functions (*CFG_coverage*). The results indicate that a significant portion (87.93%) of the control flow graph is successfully hidden, demonstrating a high obfuscation effect.

For the *call graph hiding* method, we calculated the hidden call count (*#hid_call*) and the corresponding hidden ratio (*#hid_ratio = #hid_call / #call*). Compared to C programs, C++ programs have a lower hidden ratio due to existing calls in vtable form, which were left untouched. A complementary obfuscation technique is to expand the vtable targets for these calls, such as merging different C++ class tables into a single table. We leave this as a future work.

Finally, to evaluate the impact of neither the control flow graph nor the call graph hiding methods on certain functions, we calculated the ratio of functions not obfuscated by either method (*unhidden_func_ratio*). These functions typically consist of a single basic block, have their entire function body within a loop, or contain predominantly exception-related code. The results reveal that only a minority of functions (less than 20%) remained unobfuscated using the hidden primitive.

## 6 Related Work

**Inter-procedural Transformation.** Function outlining [53] is an optimization technique to reduce binary size. Its primary objective is to identify identical code in different functions and extract it into an individual function. Consequently, all references to the extracted code are replaced with function calls. Partial inlining [70] utilizes outlining to divide code into cold and hot parts. Only the hot code of a function is inlined, while the cold code remains outlined as separate functions.

**Inter-procedural Binary Diffing.** Binary diffing works that considered inter-procedural transformation (e.g., function inlining) can be divided into three types, we discuss them as follows:

- Works that perform inter-procedural transformations that are specifically designed for function inlining while diffing. For example, BinGo [8] selectively inlines functions to imitate the inline effect. In contrast, CodeExtract [31] identifies similar code snippets and extracts them to a new function to eliminate the inline effect.
- Works that use a different granularity than function. For example, iBinHunt [44] uses deep taint and automatic input generation to find semantic differences in inter-procedural control flows. BinDiff [73] gives a similarity score in the binary level by considering the information of different granularity. InnerEye[72], BinSequence[30], and DeepBinDiff[17] try to use code fragments (e.g., basic block) with call graph as the matching granularity.

- Works that extract inter-procedural information to improve the diffing accuracy. For example, $\alpha$Diff [38] assumes the call graph is relatively stable and extracts the call graph features. DiscovRE [19] used the number of incoming calls as its filter to speed up the diffing process. Chariton et al. [33] used the algorithm in BinDiff with its specific chosen features extracted from the call graph. SIGMA [1] merged the control flow graph, register flow graph, and call graph into a joint data structure to provide a more comprehensive representation.

After the binary is obfuscated by the three obfuscation primitives in KHAOS, the extracted inter-procedural information undergoes significant changes, thereby undermining the effectiveness of these diffing methods. Regarding the incoming call count, the fusion primitive notably alters it when a callee is merged with another function. Additionally, for diffing works that operate at a smaller granularity, the reduced granularity introduces additional overhead for binary diffing. Conversely, smaller code fragments may become indistinguishable from others, thereby increasing the false positive ratio. This limitation is one of the sources of inspiration for our work.

## 7 Discussion and Future Work

Aside from obfuscation techniques, we found that existing obfuscators have limitations in their implementation. In O-LLVM [32], *Sub* can be optimized back under the LLVM O3 option, which leads us to choose O2 as our baseline. *Bog* and *Fla* skip the exception-relevant functions. For Tigress[9], we could not evaluate it in the same way as O-LLVM due to compilation errors.

The diffing process can be seen as a feature-searching process. After we separate and aggregate these features, the search difficulty increases and the search accuracy decreases. From our conclusion, the lack of call-graph consideration prevents them from adopting inter-procedural obfuscation. We hope our study will raise awareness of inter-procedural obfuscation on binary diffing.

Smaller diffing granularity brings higher diffing costs. One way to reduce the cost is to use context information to narrow the search space. Previous works pay much more attention to control flow rather than data flow. From the binary diffing perspective, data flow is harder to capture and encode. But from the obfuscation perspective, data flow is harder to change, too. Therefore, we predict the potential of data flow representation can be further tapped.

## 8 Conclusion

Binary diffing techniques can be used for 1-day/n-day vulnerability searching by attackers. In this paper, we propose an inter-procedural obfuscation technique KHAOS to protect software against state-of-the-art binary diffing. We design three obfuscation primitives — the fission, the fusion, and the hidden. Experimental results show that KHAOS is not only effective but also efficient. We wish our study could not only help developers protect their software but also promote the development of binary diffing techniques.



## Acknowledgments

This research was supported by the CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118).

## References

[1] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2015. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation* (2015).

[2] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2018. Fossil: a resilient and efficient system for identifying foss functions in malware binaries. *TOPS* (2018). https://doi.org/10.1145/3175492

[3] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *ACSAC*. https://doi.org/10.1145/2991079.2991114

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *CCS*. https://doi.org/10.1145/3133956.3134020

[5] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. https://doi.org/10.1145/2430553.2430557
[6] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. 2016. Evaluating op-code frequency histograms in malware and third-party mobile applications. In *ICETE*.
[7] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, Xiaoxue Wu, Chuanqi Tao, Tao Zhang, and Wei Liu. 2023. Learning to Detect Memory-related Vulnerabilities. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2023). https://doi.org/10.1145/3624744
[8] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *FSE*. https://doi.org/10.1145/2950290.2950350
[9] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. 2012. Distributed application tamper detection via continuous software updates. In *ACSAC*. https://doi.org/10.1145/2420950.2420997
[10] Ang Cui, Michael Costello, and Salvatore Stolfo. 2013. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*.
[11] Anderson Faustino da Silva, Edson Borin, Fernando Magno Quintao Pereira, Nilton Luiz Queiroz Junior, and Otavio Oliveira Napoli. 2022. Program representations for predictive compilation: State of affairs in the early 20's. *Journal of Computer Languages* (2022).
[12] Thaís Damásio, Michael Canesche, Vinícius Pacheco, Marcus Botacin, Anderson Faustino da Silva, and Fernando M Quintão Pereira. 2023. A Game-Based Framework to Compare Program Classifiers and Evaders. In *CGO*.
[13] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *PLDI*. https://doi.org/10.1145/3062341.3062387
[14] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices* 53, 2 (2018), 392–404. https://doi.org/10.1145/3173162.3177157
[15] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360. https://doi.org/10.1145/2594291.2594343
[16] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *S&P*. https://doi.org/10.1109/SP.2019.00003
[17] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2020.24311
[18] Thomas Dullien. [n. d.]. Searching statically-linked vulnerable library functions in executable code. https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html.
[19] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *NDSS*, Vol. 52. 58–79. https://doi.org/10.14722/ndss.2016.23185
[20] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting conditional formulas for cross-platform bug search. In *Asia CCS*. https://doi.org/10.1145/3052973.3052995
[21] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *CCS*. https://doi.org/10.1145/2976749.2978370
[22] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In *ASE*. IEEE. https://doi.org/10.1145/3238147.3240480
[23] H. Geng, M. Zhong, P. Zhang, F. Lv, and X. Feng. 2023. OPTango: Multi-central Representation Learning against Innumerable Compiler Optimization for Binary Diffing. In *ISSRE*.
[24] Artyom V Gorchakov, Liliya A Demidova, and Peter N Sovietov. 2023. Analysis of Program Representations Based on Abstract Syntax Trees and Higher-Order Markov Chains for Source Code Classification Task. *Future Internet* (2023).
[25] Sibel Gülmez and Ibrahim Sogukpinar. 2021. Graph-based malware detection using opcode sequences. In *ISDFS*.
[26] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. In *ICSE*. https://doi.org/10.1145/3180155.3180228
[27] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture binary semantics understanding via similar code comparison. In *SANER*. https://doi.org/10.1109/SANER.2016.50
[28] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *ICPC*. https://doi.org/10.1109/ICPC.2017.22
[29] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. 2018. Binmatch: A semantics-based hybrid approach on binary code clone analysis. In *ICSME*. https://doi.org/10.1109/ICSME.2018.00019
[30] He Huang, Amr M Youssef, and Mourad Debbabi. 2017. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 155–166. https://doi.org/10.1145/3052973.3052974
[31] Lichen Jia, Chenggang Wu, Peihua Zhang, and Zhe Wang. 2024. CodeExtract: Enhancing Binary Code Similarity Detection with Code Extraction Techniques. In *LCTES*.
[32] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM–software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 3–9. https://doi.org/10.1109/SPRO.2015.10

[33] Chariton Karamitas and Athanasios Kehagias. 2018. Efficient features for function matching between binary executables. In *SANER*.
[34] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2018. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Computers & Security* (2018).
[35] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 721–732. https://doi.org/10.1145/2508859.2516713
[36] Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *TOPLAS* (1979). https://doi.org/10.1145/357062.357071
[37] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *ISSTA*.
[38] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. $\alpha$diff: cross-version binary code similarity detection with dnn. In *ASE*. https://doi.org/10.1145/3238147.3238199
[39] LLVM Project. 2022. LLVM Block Frequency Terminology. https://llvm.org/docs/BlockFrequencyTerminology.html.
[40] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE*. https://doi.org/10.1145/2635868.2635900
[41] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.. In *NDSS*.
[42] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *USENIX Security*.
[43] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *DIMVA*. Springer. https://doi.org/10.1007/978-3-030-22038-9_15
[44] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *ICISC*.
[45] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. {BinSim}: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *USENIX Security*.
[46] Carey Nachenberg. 1997. Computer virus-antivirus coevolution. *Commun. ACM* 40, 1 (1997), 46–51. https://doi.org/10.1145/242857.242869
[47] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *S&P*. IEEE. https://doi.org/10.1109/SP.2015.48
[48] Oreans Technologies. 2022. Themida Overview. https://www.oreans.com/themida.php.
[49] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2021. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. arXiv:2012.08680 [cs.CR] https://arxiv.org/abs/2012.08680
[50] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 709–724. https://doi.org/10.1109/SP.2015.49
[51] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. 2004. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *ICSM*. https://doi.org/10.1109/ICSM.2004.1357803
[52] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *PLDI*. https://doi.org/10.1145/3453483.3454035
[53] River Riddle. [n. d.]. Interprocedural IR Outlining For Code Size. https://llvm.org/devmtg/2017-10/slides/Riddle-Interprocedural%20IR%20Outlining%20For%20Code%20Size.pdf.
[54] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *CSUR* (2016). https://doi.org/10.1145/2886012
[55] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P*. https://doi.org/10.1109/SP.2015.51
[56] Noam Shalev and Nimrod Partush. 2018. Binary Similarity Detection Using Machine Learning. In *PLAS*.
[57] Mechthild Stoer and Frank Wagner. 1997. A simple min-cut algorithm. *JACM* (1997). https://doi.org/10.1145/263867.263872
[58] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *S&P*. https://doi.org/10.1109/SP.2015.46
[59] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. JTrans: Jump-Aware Transformer for Binary Code Similarity Detection. In *ISSTA*.
[60] Huaijin Wang, Shuai Wang, Dongpeng Xu, Xiangyu Zhang, and Xiao Liu. 2020. Generating effective software obfuscation sequences with reinforcement learning. *IEEE Transactions on Dependable and Secure Computing* (2020). https://doi.org/10.1109/TDSC.2020.3041655
[61] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *ASE*. https://doi.org/10.1109/ASE.2017.8115645
[62] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *ASE*. https://doi.org/10.1145/2970276.2970312

[63] xorpd. [n. d.]. FCatalog. https://www.xorpd.net/pages/fcatalog.html.
[64] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. 2018. Manufacturing resilient bi-opaque predicates against symbolic execution. In *DSN*. https://doi.org/10.1109/DSN.2018.00073
[65] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In *ISSTA*.
[66] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-enabled Software Reuse Detection Based on a Multi-Level Birthmark Model. In *ICSE*. https://doi.org/10.1109/ICSE43902.2021.00084
[67] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering* 45, 11 (2018), 1125–1149. https://doi.org/10.1109/TSE.2018.2827379
[68] Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. 2023. Asteria-Pro: Enhancing Deep Learning-based Binary Code Similarity Detection by Incorporating Domain Knowledge. *ACM TOSEM* (2023). https://doi.org/10.1145/3604611
[69] Peihua Zhang, Chenggang Wu, Mingfan Peng, Kai Zeng, Ding Yu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2023. Khaos: The Impact of Inter-procedural Code Obfuscation on Binary Diffing Techniques. In *CGO*.
[70] Peng Zhao and J.N. Amaral. 2005. Function outlining and partial inlining. In *SBAC-PAD*.
[71] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Jianjun Chen, Zhijian Ou, Min Yang, and Chao Zhang. 2023. Callee: Recovering Call Graphs for Binaries with Transfer and Contrastive Learning. In *S&P*. https://doi.org/10.1109/SP46215.2023.10179482
[72] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *NDSS*. https://doi.org/10.14722/ndss.2019.23492
[73] zynamics GmbH and Google LLC. 2022. BinDiff Manual. http://www.zynamics.com/bindiff/manual/index.html.

Received 7 November 2023; revised 13 July 2024; accepted 9 October 2024