KHAOS: The Impact of Inter-procedural Code Obfuscation on Binary Diffing Techniques

Peihua Zhang*
Chenggang Wu*†
Mingfan Peng*
zhangpeihua@ict.ac.cn
wucg@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
Beijing, China

Kai Zeng*
Ding Yu*
Yuanming Lai*‡
laiyuanming@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
Beijing, China

Yan Kang*
Wei Wang
Zhe Wang†
wangzhe12@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
Beijing, China

Abstract

Software obfuscation techniques can prevent binary diffing techniques from locating vulnerable code by obfuscating the third-party code, to achieve the purpose of protecting embedded device software. With the rapid development of binary diffing techniques, they can achieve more and more accurate function matching and identification by extracting the features within the function. This makes existing software obfuscation techniques, which mainly focus on the intra-procedural code obfuscation, no longer effective.

In this paper, we propose a new inter-procedural code obfuscation mechanism KHAOS, which moves the code across functions to obfuscate the function by using compilation optimizations. Two obfuscation primitives are proposed to separate and aggregate the function, which are called fission and fusion respectively. A prototype of KHAOS is implemented based on the LLVM compiler and evaluated on a large number of real-world programs including SPEC CPU 2006 & 2017, CoreUtils, JavaScript engines, etc. Experimental results show that KHAOS outperforms existing code obfuscations and can significantly reduce the accuracy rates of five state-of-the-art binary diffing techniques (less than 19%) with lower runtime overhead (less than 7%).

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

*Also with University of Chinese Academy of Sciences.
†Also with Zhongguancun Laboratory.
‡Yuanming Lai is the corresponding author.

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0101-6/23/02.
https://doi.org/10.1145/3579990.3580007

Keywords: Software Protection, Obfuscation, Binary Diffing

ACM Reference Format:
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 *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579990.3580007

1 Introduction

Embedded devices have been widespread in many fields of modern life, such as wearables, traffic lights, and autonomous driving vision sensors and the total number are expected to reach 30 billion by 2025 [47]. In recent years, the number of vulnerabilities disclosed in embedded device software has been on the rise, and attacks targeting embedded devices have increased more than fivefold in the past four years [35]. Once a vulnerability in an embedded device is exploited, it can lead to the collapse of the backbone network [4], while vulnerabilities in medical devices such as pacemakers are life-threatening [8, 38].

In addition to directly writing flawed code to introduce vulnerabilities, the reuse of vulnerable third-party code is another important reason for the widespread existence of vulnerabilities in embedded devices. For example, Cui et al. [11] found that 80.4% of LaserJet printers used third-party libraries with known vulnerabilities. However, vulnerabilities in these embedded devices cannot be patched in time due to the fragment issues — similar code exists in multiple versions of various products due to the fast replacement of embedded devices [56]. For example, the QualPwn vulnerability [49] in the Qualcomm's WiFi controller, which is equipped in millions of Android phones, took nearly 6 months from the vulnerability disclose to the patch released by the Qualcomm, and OEMs took longer to patch all devices across all versions.

Unfortunately, the above problem favors attackers in which they could 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 [3, 7, 12, 13, 16, 17, 19–26, 33, 36, 52, 53, 55, 59, 60, 62] to locate the vulnerable code reused in the binary by comparing the binary with the third-party code. With the introduction of machine learning, binary differing techniques have made great progress in recent years. This greatly facilitates attackers locating existing vulnerabilities in binaries. For example, David et al. [13] searched for common vulnerabilities in mobile devices, wearable devices, and medical devices, and were able to locate 373 existing vulnerabilities.

Software obfuscation techniques [5, 9, 27, 54, 57] can transform the program code to change the characteristics of the binary code. They could be used against binary diffing techniques, preventing attackers from locating existing vulnerabilities and thus protecting software. Recent researches have shown that software obfuscation techniques are no longer effective against the state-of-the-art binary diffing techniques [16, 33, 36, 55]. The main reason is that most software obfuscation techniques focus on the intra-procedural code 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 their ability to change function semantics at the binary level. To this end, we propose an inter-procedural code 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 and aggregating functions into one.

It is non-trivial that transform arbitrary functions in KHAOS due to the challenges posed by performance, correctness, and obfuscation effect. For example, 1) To balance the obfuscation effect with the performance overhead, choosing which code blocks within a function (or functions) to be separated (or aggregated) is a problem; 2) Rebuilding all control flow and data flow among functions after transformations (especially the indirect function calls handling in the fusion) is difficult; 3) Aggregating functions deeply without affecting the functionality of each function is a problem.

To address these challenges, two obfuscation primitives are proposed in KHAOS— the *fission* primitive and the *fusion* primitive. The fission is used to separate a function into sub-functions, and the fusion is used to aggregate several functions into one. The fission and the fusion are two complementary primitives, in which the fission tries to obfuscate the function by itself and the fusion tries to obfuscate the function by other functions. Furthermore, these two primitives can also be used together to improve the obfuscation effect, that is, the sub-functions separated by the fission can be aggregated with other functions again.

The fission partitions the code region to a sub-function on the control flow of the function with the dominator tree as the granularity, and also combines the static cold/hot code analysis technique to achieve lower performance overhead. Since the define-use relationships of variables are changed from within a function to cross functions, the fission needs to rebuild the data flow by passing parameters. To minimize the performance degradation caused by parameters passing, we also propose a data-flow reduction mechanism to reduce the number of parameters of the sub-functions. The control flow (including the exception 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 selects two functions with compatible return values and no variadic parameters for the aggregation. Compatible means converting between different data types without losing precision. The parameter list of the post-aggregation function is merged from these two functions. To avoid the inefficient way of passing parameters through the stack, we propose a parameter list compression mechanism to reduce the number of the 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. We also propose a trampoline mechanism to handle the function calls across modules. To further improve the obfuscation effect, the deep fusion method is proposed to aggregate innocuous basic blocks, whose execution does not affect the global memory state, from different functions together within the aggregated function.

KHAOS was implemented based on the LLVM framework. The experimental evaluations were conducted on the Linux/X86_64 platform by using SPEC CPU 2006 & 2017 C/C++ programs, CoreUtils, and 5 common embedded device software containing vulnerabilities. Five state-of-the-art binary diffing tools [16, 17, 22, 34, 62] 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 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 are as follows:

- **A novel inter-procedural code obfuscation mechanism.** We point out that the inter-procedural code obfuscation is necessary against the binary diffing techniques, and propose a new obfuscation mechanism, KHAOS, which could obfuscate the code across functions.

- **The fission and the fusion primitives.** We propose two obfuscation primitives in KHAOS to move the code across functions. The fission separates a function into multiple functions, and the fusion aggregates multiple functions into one.
- **New insights from implementation and evaluation.** We implement and evaluate a prototype of KHAOS, and the results show that it outperforms the existing obfuscators against the state-of-the-art binary diffing techniques. Our study suggests that binary diffing techniques should focus more on extracting the inter-procedural code features.

2 Background and Motivation

2.1 Binary Diffing

Binary diffing is a technique for visualizing and identifying differences between two binaries. It can quantitatively measure the differences between binaries and give matching result at predefined granularity (e.g., function). It has been widely used in software vulnerability search, security patch analysis, malware detection, code clone detection, etc.

The workflow of binary diffing can be divided into two stages, i.e., the offline features extraction and the online code search. On the offline stage, tools extract features from binaries, while what features should be extracted is the focus of recent research; On the online stage, tools calculate the similarity of the given binaries by using extracted features. For example, BinDiff [62] extracts the number of basic blocks, control flow edges, and function calls within a function as the function's identity. Then, it combines the control flow graph matching algorithm to search for similar functions.

2.2 Software Obfuscation

Software obfuscation transforms the program without changing its functionality to make it hard to be analyzed. It can be used to hide vulnerabilities, protect intellectual property, etc. Actually, there is an arm 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. In recent decades, there are various techniques proposed in software obfuscation, and they can be classified into data obfuscation, static code rewriting, and dynamic code rewriting [45].

Data obfuscation techniques [10] transform the format of data to prevent it from direct matching. Since most binary diffing techniques utilize the features of the code, obfuscating data is less effective against binary diffing.

Various dynamic code rewriting approaches follow the concept of packing [39, 44], which hides code by encoding or encrypting it as data. But, the packing techniques are easy to be automatically unpacked [1, 41] or be memory-dumped [15, 18, 46, 51], which would lose the effect of obfuscation. Code virtualization is another popular obfuscation technique [28]. It translates code into specific interpret representations (IRs) instead of the native instructions and then uses an engine to interpret the IRs at runtime. This technique sacrifices much performance (10x slowdown at least [29]) 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, 27] replaces the original instruction with equivalent instruction(s), such as replacing an "`add`" instruction with two "`sub`" instructions. O-LLVM [27] designed 10 different substitution strategies for arithmetic and logical operations. To increase the complexity of conditional branch instructions, opaque predicate techniques [9, 27, 37, 50, 57] were proposed. They add permanent true or false (e.g., x^2 != -1) conditions that do not affect the original control flow, which are frequently used against analytical techniques such as symbolic execution.

Basic block level: Bogus control flow [9, 27, 40] 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 program's functionality.

Function level: Control flow flattening [9, 27] converts the control flow of the function into the "`switch-case`" form, which is hard to be analyzed, 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).

2.3 Motivation

As binary diffing techniques continue to advance, many static code rewriting techniques (referred to as *code obfuscation* in the rest of the paper) with the intra-procedural granularity (i.e., instruction, basic block, and function) are no longer effective [16, 33, 36, 55]. The main reason is that intra-procedure code obfuscations do not fundamentally change the semantics of each function, while most binary diffing techniques are increasingly capable of extracting features within functions to understand their semantics.

Therefore, we argue that *inter-procedural code obfuscations should be emphasized due to their ability to change function semantics at the binary level which is the key to defeating binary diffing techniques*: 1) For the binary diffing works that only consider the intra-procedural information, the inter-procedural code obfuscation can fundamentally defeat them because the code structures along with the semantics are significantly changed; 2) For the binary diffing works that take inter-procedure information into account, the inter-procedural code obfuscation can also defeat them because

the inter-procedural information extracted, such as the types of function calls, the numbers of function calls, and the call graph, are also significantly changed after the obfuscation.

Our thinking is also hinted by the literature published from both the offensive and defensive sides: 1) most of binary diffing works have discussed the issues of function inline [3, 6, 7, 12, 14, 16, 19–21, 23–26, 33, 58], and many of them [3, 13, 14, 19–21, 24–26, 58] admitted that it would affect the accuracy of diffing; 2) Ren et al. [43] found the function inline could reduce the binary similarity by approximately 10%.

3 Our Solution: Khaos

3.1 Overview

To achieve the inter-procedural code obfuscation, Khaos changes the amount of code within a function by moving code across functions firstly and then utilizes the compiler's optimizations to transform (obfuscate) the code. The idea behind it is that *once the code is restructured among functions, the generated binary code after compilation optimizations (especially intra-procedural optimizations) can be very different.*

In detail, we propose two obfuscation primitives — the *fission* primitive and the *fusion* primitive. The 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. These two primitives can also be used together to make more in-depth changes to the function, that is, the separated sub-functions can be aggregated with other functions.

For the convenience of discussion, we denote a function before the transformation as an **oriFunc** (short for *original function*), and denote the new function formed after the fusion as the **fusFunc** (short for *fused function*). The new function formed by the separated code during the fission is denoted as the **sepFunc** (short for *separated function*), and the function formed by the remaining code is denoted as the **remFunc** (short for *remnant function*).

Fig. 1 gives an example about how the fission and the fusion are performed on a function named `cal_file()`. The function is used to find the number of a special character in a given file. It first checks the file name and open the file (lines 4-7), then reads the content and counts the amount (lines 9-11). We can see that the fission separates two basic blocks (②③) to `sepFunc-1`, and four basic blocks (⑤-⑧) to `sepFunc-2`, respectively. To maintain the correctness, the fission inserts three trampoline basic blocks in the `remFunc-1` (ⓐⓑⓒ) to create the call relationship of two *sepFunc*s. Basic block (ⓓ) is used to return different value of `sepFunc-1` (detailed in §3.2). On top of the fission, the fusion aggregates the `log()` function and the `sepFunc-2` into a `fusFunc-1`. The entry basic block (ⓔ) will be inserted into the `fusFunc-1` to select the aggregated code blocks.

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

- **Challenge-1:** Choosing which basic blocks (or functions) to be separated (or aggregated) 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 brings 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:** Simply merging functions makes they become each other's junk code and has a limited obfuscation effect because the compiler will still optimize the code for different functions separately. Binding control flows and data flows belonging to different functions in the *fusFunc* can prevent that but is also challenging to avoid changing the functionality of the function.

In the following subsections, we will detail the fission and the fusion design, and how we address the above challenges.

3.2 The Fission Primitive

The fission first identifies the regions (each region is a basic block set) that need to be separated, then composes these regions into *sepFunc*s, and finally rebuilds the control flow and the data flow among *sepFunc*s and *remFunc*.

3.2.1 Partitioning Regions to Form SepFunc. In general, a function's property is 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 [2] on the control flow graph, it can be extracted into a *sepFunc*. The fission creates call relationship 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 identify algorithm is the key to reducing the overhead and improving the obfuscation effect.

The core idea. 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.

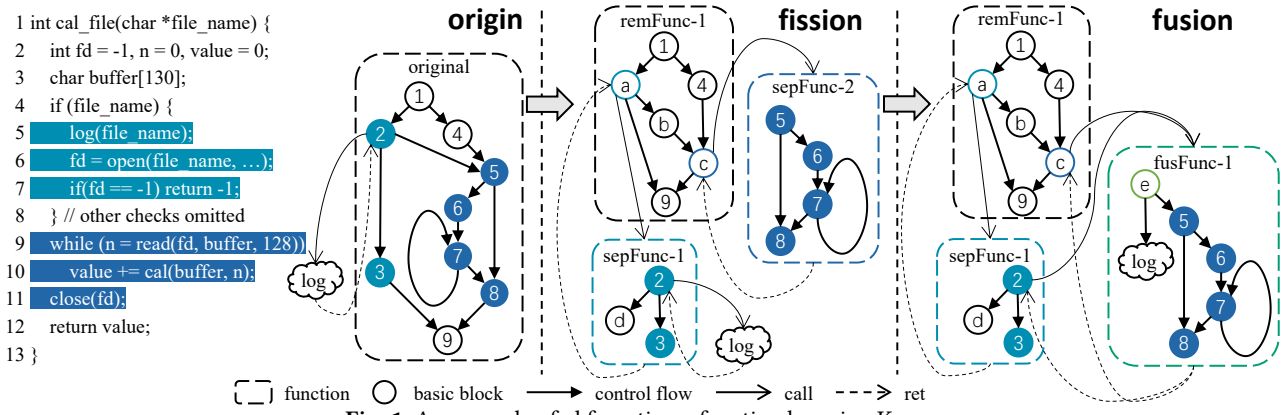

Fig. 1. An example of obfuscating a function by using KHAOS.

Algorithm 1 The region identifying algorithm

```
1:  procedure IDENTIFY(f) ▷ The function's representation
2:      get dominator tree set S of f
3:      S ← S \ f  ▷ We won't separate the whole function
4:      while S is not empty do
5:          target ← null
6:          for dominator tree t in S do
7:              effect ← basic block count of t
8:              cost ← frequency of t's head
9:              if t is in loop then
10:                 l ← the innermost loop where t is located
11:                 cost ← loop count of l × cost
12:             end if
13:             value ← effect ÷ cost
14:             if value > target's value then
15:                 target ← t      ▷ Update the chosen tree
16:             end if
17:         end for
18:         if target ≠ null then
19:             delete trees from S that intersect with target
20:         end if
21:     end while
22:     return
23: end procedure
```

The region identifying algorithm. Based on the above idea, we design the region identifying algorithm (algorithm 1) on top of the directed weighted graph cut algorithm [48] to balance the performance overhead and the obfuscation effect. The algorithm takes function code as input and performs dominator tree analysis [31] (line 2) at first. To avoid separating the whole function body into a *sepFunc*, we remove the dominator tree of function itself (line 3) and identify the regions from the rest of the trees. To indicate the effect of the fission on obfuscation, we use the number of basic blocks in the tree to represent it (line 7). 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 [32] (line 8) and the loop count (if the region is in a loop, the call to *sepFunc* will increase) as the cost of the cut (lines 8-12). We iteratively select the most cost-effective (i.e., maximum the ratio of effect and cost) dominator tree to separate until the tree set is empty (lines 13-16).

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 return value of *sepFunc*s. For each variable used in a region, it should be an input if its point is outside the region; Similarly, for each variable defined in a region, it should be an 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 relationship are across 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

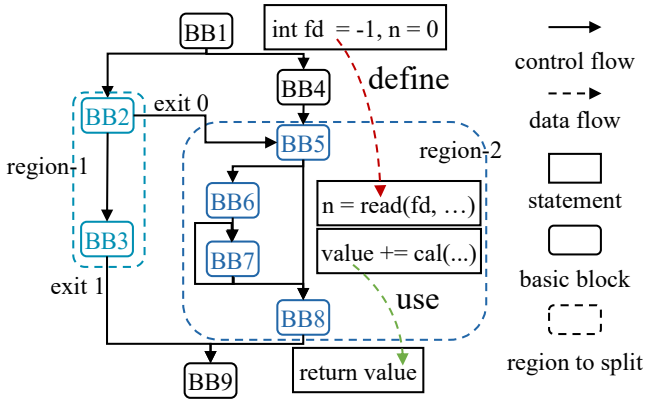

Fig. 2. The control-flow and data-flow graphs of cal_file() in Fig. 1

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*.

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*, uses 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 (EH in short). The fission requires special handles of them.

Handling the setjmp/longjmp. Programmers could use the `setjmp()` in a function 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 place the `jmpbuf` is pointing to, i.e., the call-site of the `setjmp()`. 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 calling the `setjmp()` cannot be freed when the corresponding `longjmp()` is executed. Otherwise, the `longjmp()` will direct control flow to an unknown location.

Handling the C++ exception. The EH mechanism is a feature of the C++ that developers can capture exceptions in the `try` block by writing the `catch` statements. Since the fission moves part of the code into a *sepFunc*, the `try-catch` pair may be broken, making EH information inconsistent. Simply skipping the exception-relevant function 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 same region.

3.3 The Fusion Primitive

The fusion selects functions to form *fusFunc*, and rebuilds the control and the data flow to ensure the correctness. 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 the 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, there are other conditions that 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 a 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.

Parameter list compression. Simply merging the parameter lists of the two functions makes the parameter list of *fusFunc* too long, which will degrade the performance of calling *fusFunc*. This is because in the X86_64 calling convention, the first six parameters are passed in registers, and the rest of the parameters are 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

```
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.

the `foo()` have an integer parameter (`short a` and `int m`), we compress them into one integer parameter (`int x`).

If a parameter can not participate in the compression, it is copied into the parameter list of the *fusFunc*. The number of parameters after the fusion will increase. In the worst case, it 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 the 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 need to be removed, and all call-sites to the *oriFunc*s need to be replaced to call the *fusFunc*. As mentioned before, a `ctrl` parameter will be added into the parameter list of 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`.

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*. But, we encounter a problem that we do not know what the value of the `ctrl` parameter should be set to. This is because at the

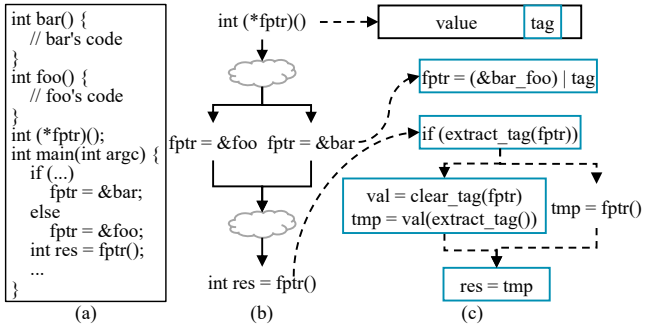

Fig. 4. Function reference and indirect call processing.

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 [30]. 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. In detail, 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 as the and set the `ctrl` parameter according to the *tag*.

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, as shown in Fig. 4 (c), if the pointer `fptr` points to the `bar()`, the value of the *tag* will be set to `11b`. When the pointer `fptr` is dereferenced to make a call, we insert code to first 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.

We choose the 2nd bit and the 3rd bit of function pointers to place the *tag*. This is because the functions are usually

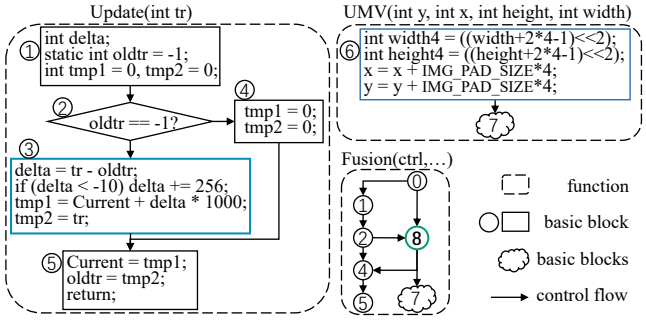

Fig. 5. A real-world example of the deep fusion method.

16-bytes aligned with the performance consideration, so the lowest 4 bits of the function pointer can be used.

Handling function calls across modules. There are two cases of cross-module function calls, one is the function pointer of a module is propagated to other modules, and the other is a module directly calls functions exported by other modules. If any case happens on a *fusFunc*, we needs to process all involved modules to ensure the *fusFunc* can be called correctly. But in some cases, we can not process all the modules (e.g., some libraries may have no source code).

To address this problem, we propose a trampoline mechanism so that all modules do not need to be processed. In detail, we transverse the data flow conservatively and identify all function pointers that may propagate outside the module. And then, we modify these function pointers to point to a piece of trampoline code instead of the *fusFunc*. So that when the external module calls these function pointers, the control flow will transfer to the trampoline code first, and the trampoline code will help the function outside the module to reorganize the function parameters and call the *fusFunc*. For the exported *oriFunc*s, the method is similar to the replacing the *oriFunc*'s function body with the trampoline code.

3.3.4 The Deep Fusion.

To further improve the obfuscation effect, we propose a deep fusion method to aggregate as many basic blocks as possible between the two parts of the code during the fusion process.

We have observed that *some basic blocks can be executed many times without affecting the normal function*. The characteristic of these basic blocks is that their execution does not affect the global memory state, and they are called the *innocuous basic block* in this paper. The concept is very similar to the *reentrant function* [42] that it can be re-executed without affecting the functionality of the program. For innocuous basic blocks from different *oriFunc*s, they can be aggregated together within the *fusFunc*. The innocuous analysis of each basic block is conservative. For example, 1) if a memory write operation in a basic block cannot be determined whether the modified data is local or global, then this basic block is not innocuous; 2) if there is a function call to an external function in a basic block, this basic block is not innocuous.

We give a simplified example of `464.h264ref` in SPEC CPU 2006 benchmark. As shown in Fig. 5, the `Update()` and `UMV()` are aggregated into the `Fusion()`. The basic block (BB) ③ of the `Update()` firstly redefines the local variable `delta`, and then loads the value of global variable `Current`, and writes two local variables `tmp1` and `tmp2` at last. Since these operations do not affect the global memory state, the BB ③ is determined to be innocuous, and so as the BB ⑥ of UMV(), thus we aggregate them into one — the BB ⑧.

This deep fusion method modifies the control flow graph and data flow graph of the *fusFunc* at the same time, adding data dependency and control dependency so that the *fusFunc* cannot be simply separated back to the two functions.

3.4 Combining the Fission and the Fusion

The fission and the fusion can be used together to further enhance the obfuscation effect.

- **FuFi.sep**: Only aggregating the *sepFunc*s generated by the fission. In this case, the issue of handling indirect function calls no longer exists;
- **FuFi.ori**: Only aggregating the *oriFunc*s that are not processed by the fission, e.g., the functions with only one basic block. This combination could balance the obfuscation effect and the performance overhead well, and is suitable for software in most real-world scenarios;
- **FuFi.all**: Aggregating the fission-generated *sepFunc*s and the fission-unprocessed *oriFunc*s uniformly and randomly. In this combination, the obfuscation effect is prioritized, followed by the performance overhead. It is suitable for programs that require a high obfuscation effect.

4 Evaluation

We implemented KHAOS based on the LLVM-9.0.1. The fission and the fusion are implemented as the middle-end passes, and the fission pass is scheduled before the fusion pass. We run KHAOS on Ubuntu 20.04 (Kernel v5.4.0) that runs on an Intel(R) Xeon(R) Gold 5218 CPU with 128G 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?

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 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

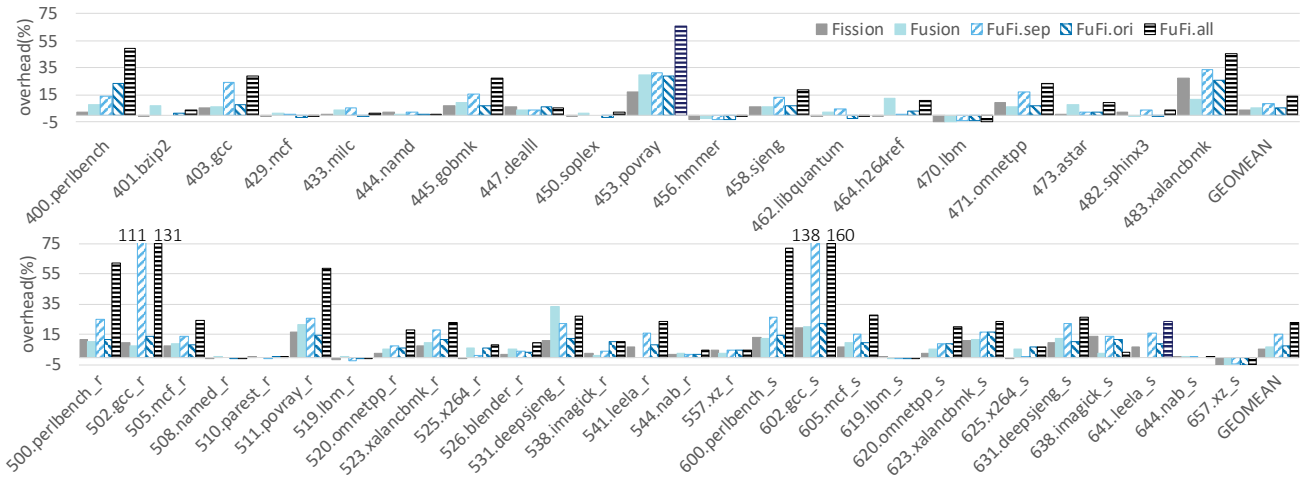

Fig. 6. Runtime overhead of SPEC CPU 2006 (upper part) and 2017 (lower part) C/C++ programs.

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 (LTO).

Comparison targets. To compare with existing obfuscator, we choose the popular compiler-level obfuscation tool O-LLVM [27] as our comparison target because it is open-sourced and compiler-based (same as KHAOS). O-LLVM [27] contains three obfuscation methods: instruction substitution (*Sub*), bogus control flow (*Bog*), and control flow flattening (*Fla*). Literatures [5, 16, 43, 55] in software engineering, systems security, and programming languages fields all use it in their experiments. To ensure the consistency of the evaluation environment, we upgrade the LLVM version of O-LLVM [27] to 9.0.1, which is same as KHAOS. We also choose BinTuner [43], 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 compiler's options.

Confrontation targets. We use five state-of-the-art binary diffing techniques, i.e., Google BinDiff [62], VulSeeker [22], Asm2Vec [16], SAFE [34], DeepBinDiff [17], to evaluate the effectiveness of KHAOS. Among them, Google BinDiff is an industry-standard binary diffing tool. Asm2Vec, SAFE, DeepBinDiff, and VulSeeker are the state-of-the-art methods for learning the semantic similarity in different granularity (e.g., function, basic block, control flow graph, call graph).

4.1 Performance Overhead After Obfuscation

We separately evaluated the performance overhead of the fission and the fusion, and the three combination modes introduced in §3.4 on the **T-I**. As shown in Fig. 6, the geometric performance overhead of the fission and the fusion are 5% and 6%, 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 improves the code locality of the aggregated functions. The results demonstrated that obfuscations compliant with the compiler optimizations can have good performance advantages.

Compared with the *FuFi.ori*, the other two combinations have a higher overhead because the fission generates many *sepFuncs*, aggregating them all incurs non-negligible performance overhead. For example, the `502.gcc_r` contains many recursive functions, the *sepFunc*s generated by these functions are aggregated to the *fusFunc*s which are also the recursive functions. Since the stack frames of *fusFunc*s are larger, they will bring much pressure to the stack.

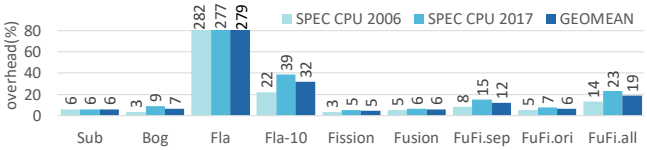

Fig. 7. Runtime overhead of O-LLVM and KHAOS.

Compared with O-LLVM[27] and compiler options. We compared the performance overhead of KHAOS with *Sub*, *Bog*, *Fla*. As shown in Fig. 7, 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%. Due to the page limit, we put the compiler option relevant evaluation in our long version of this paper, which is available at http://arxiv.org/abs/2301.11586.

4.2 The Effectiveness Against Binary Diffing

Comparing binary diffing works is challenging due to their measurements of similarity are very different [43], such as graph edit distance or statistical significance. Simply comparing their similarity scores does not provide accurate information. For the commercial binary diffing tool BinDiff [62], we normalized its similarity score to `[0, 1]`. For other tools open-sourced in academia, we normalized their results by

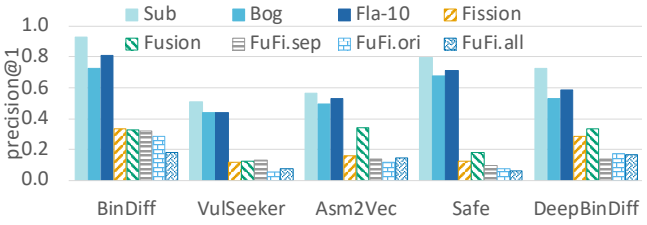

Fig. 8. Precision@1 result of chosen binary diffing work.

computing the ratio of true matching function pairs that are also the top-ranked matching candidates (i.e. *Precision@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 DeepBinDiff [17], since its result is basic block to 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 adaptability.

The test suites for VulSeeker [22] and DeepBinDiff [17] need to be adjusted due to unable to run results. VulSeeker [22] takes more than 1 day to diff two large binaries and often gets killed due to memory limit. To speed up VulSeeker, we group the related functions into small groups (30 functions per group) to manually reduce the searching space, which is unfavorable to KHAOS because the smaller the group size, the easier to diff. DeepBinDiff [17] 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 decide not to modify it and only use programs 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 is also 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 Fig. 8 shows, higher accuracy means lower adversarial effect. Since BinDiff [62] takes the advantage of function names, its result is much higher than others. Although DeepBinDiff [17] 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 have been changed by KHAOS, and that's why KHAOS can defeat it. With comparable overhead, KHAOS can achieve a much better adversarial effect than O-LLVM [27].

4.3 The Ability of Hiding Vulnerable Code

We use the **T-III** to further evaluate the ability of hiding real world vulnerable code. Each program contains at least one vulnerability. In this experiment, we only used VulSeeker [22], Asm2Vec [16], and SAFE [34] to calculate the *escape@n ratio* (the rank of truly matched pair in the matched result) of vulnerable functions. The reason why BinDiff and DeepBinDiff were not used is that they only give top-1 matched result. We calculated *escape@1/10/50 ratio* of vulnerable functions. For example, as shown in Fig. 9, the *escape@50 ratio* of *FuFi.all* on Asm2Vec is over 0.8, which means more than 80% of vulnerable functions can not be found within top-50 ranked functions. Moreover, this time we set the obfuscation ratio of *Fla* in O-LLVM to 100%, which would bring unacceptable overhead in the real scenario.

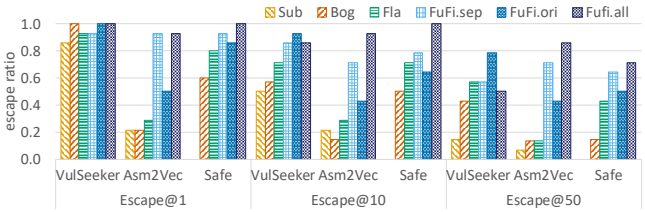

Fig. 9. Escape ratio for top@1/10/50 of vulnerable functions.

The *escape ratio* could reflect the ability of hiding the vulnerable code with different obfuscations. With the same precision and binary diffing tool (e.g., `escape@50-Asm2Vec`), the *FuFi.sep* and the *FuFi.all* are better than the *FuFi.ori*, and all of them are better than the *Sub*, the *Bog*, and the *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@50-FuFi.all`, Asm2Vec is more accurate than Safe, and both of them outperform VulSeeker. 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 Conclusion

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

Acknowledgments

This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 61902374, 62272442, U1736208, 61872386, and the Innovation Funding of ICT, CAS under Grant No.E161040.

Table 1. Vulnerable functions of Test Suite III

Program	Function	CVE
JerryScript	opfunc_spread_arguments	2020-13991
QuickJS	compute_stack_size_rec	2020-22876
BusyBox1.33.1	getvar_s	2021-42382
	handle_special	2021-42384
OpenSSL 1.1.1	init_sig_algs	2021-3449
	EC_GROUP_set_generator	2019-1547
libcurl 7.34.0	suboption	2021-22925,2021-22898
	init_wc_data	2020-8285
	conn_is_conn	2020-8231
	tftp_connect	2019-5482,2019-5436
	ftp_state_list	2018-1000120
	alloc_addbyter	2016-8618
	Curl_cookie_getlist	2016-8623
	ConnectionExists	2016-8616,2016-0755, 2014-0138,2015-3143
Total	14	19

Data-Availability Statement

We have published the diffing files [61] used in our evaluation process, along with the scripts to parse them, and the result in the paper. These diffing files are generated by the 5 diffing works mentioned in the §4.

A Artifact Appendix

A.1 Abstract

We provide all the diffing files in our artifact, along with scripts to regenerate the data for the graphs in the paper. Since SPEC CPU 2006 & 2017 are not open-sourced, we omitted the files of the performance part.

A.2 Artifact Check-List (Meta-Information)

- **Run-time environment:** Ubuntu 20.04 or 22.04.
- **Hardware:** We use Intel(R) Xeon(R) Gold 5218 and 6148 CPU for performance experiment. A similar CPU should give comparable results.
- **Execution:** Python scripts.
- **Output:** Performance overhead needs to be recalculated manually. Diffing and CVE results are output in CSV files.
- **How much disk space required?:** 20GB.
- **Publicly available?:** Yes.
- **Dataset:** Our data sets include SPEC CPU 2006, SPEC CPU 2017, and 5 programs in Table 1.
- **How delivered:** The diffing files, scripts, and results are available at https://doi.org/10.5281/zenodo.7496594.

A.3 Experiment Workflow

A.3.1 Performance Part. We put all the performance results of our platform in the *result/spec-result* folder. The detailed results are in the *SPEC CPU 2006* sheet and *SPEC CPU 2017* sheet of *result/result.xlsx*. The *overhead* sheet and the *overhead(2)* sheet summarize the detailed result and compared result, corresponding to Fig. 6 and Fig. 7, respectively.

A.3.2 Precision@1 Part. The diffing files of 5 diffing works are in the *diffing* folder. The detail of precision@1 is in *precision_1* sheet of *result/result.xlsx*. We provide a collecting script to summarize the result. This script outputs the diffing result of every tool in separated CSV files (e.g., *result/asm2vec-precision-1.csv* is corresponding to the *Asm2Vec* part of *precision_1* sheet in *result.xlsx*). The *precision_1_simplified* sheet (corresponding to the Fig. 8) in *result.xlsx* is the simplified result of *precision_1* sheet and can be re-calculated by it.

```
$ cd scripts && ./parse_precision_1.sh
```

A.3.3 BinTuner Part. The diffing files of BinDiff for the binary files generated by BinTuner is in the *diffing/BinDiff/BinTuner* folder. We provide a collecting script to summarize the result. This script outputs the diffing result BinDiff in a CSV files (*result/bintuner.csv*, corresponding to the *bintuner* sheet in *result.xlsx*). The overall runtime is about 5 seconds. The *bintuner* sheet is corresponding to the **??**.

```
$ cd scripts && ./parse_bintuner.sh
```

A.3.4 CVE Part. The detail of CVE is in *cves* sheet of *result/result.xlsx*. We provide a collecting script to summarize the result. This script outputs the CVE ranking result of every tool in separated CSV files (e.g., *result/asm2vec-cve.csv* is corresponding to the *Asm2Vec* part of *cves* sheet in *result.xlsx*). The *cve* sheet is a simplified version and is corresponding to the Fig. 9.

```
$ cd scripts && ./parse_cve.sh
```

A.4 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] Markus F.X.J. Oberhumer and László Molnár and John F. Reiser. 2022. The Ultimate Packer for eXecutables. https://upx.github.io/.

[2] Robert B Allan and Renu Laskar. 1978. On domination and independent domination numbers of a graph. *Discrete mathematics* 23, 2 (1978), 73–76. https://doi.org/10.1016/0012-365X(78)90105-X

[3] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2018. Fossil: a resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)* 21, 2 (2018), 1–34. https://doi.org/10.1145/3175492

[4] Manos Antonakakis et al. 2017. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*. 1093–1110.

[5] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 189–200. https://doi.org/10.1145/2991079.2991114

[6] 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. 1–10. https://doi.org/10.1145/2430553.2430557

[7] 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 *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689. https://doi.org/10.1145/2950290.2950350

[8] Zoe Chen, Paul O'Donnell, Eric Ottman, Steven Trieu, and Alan J Michaels. 2020. An Invisible Insider Threat: The Risks of Implanted Medical Devices in Secure Spaces. (2020).

[9] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. 2012. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 319–328. https://doi.org/10.1145/2420950.2420997

[10] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Breaking abstractions and unstructuring data structures. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225)*. IEEE, 28–38. https://doi.org/10.1109/ICCL.1998.674154

[11] Ang Cui, Michael Costello, and Salvatore Stolfo. 2013. When firmware modifications attack: A case study of embedded exploitation. (2013). https://doi.org/10.7916/D8P55NKB

[12] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 79–94. https://doi.org/10.1145/3062341.3062387

[13] 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

[14] 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

[15] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*. 51–62. https://doi.org/10.1145/1455770.1455779

[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 *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489. 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] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)* 44, 2 (2008), 1–42. https://doi.org/10.1145/2089125.2089126

[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 *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 346–359. 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491. 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 *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 896–899. https://doi.org/10.1145/3238147.3240480

[23] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture binary semantics understanding via similar code comparison. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 57–67. https://doi.org/10.1109/SANER.2016.50

[24] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 88–98. https://doi.org/10.1109/ICPC.2017.22

[25] 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 *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 104–114. https://doi.org/10.1109/ICSME.2018.00019

[26] 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

[27] 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

[28] Samuel T King and Peter M Chen. 2006. SubVirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 14–pp. https://doi.org/10.1109/SP.2006.38

[29] 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* 74 (2018), 202–220. https://doi.org/10.1016/j.cose.2018.01.008

[30] 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

[31] Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 1 (1979), 121–141. https://doi.org/10.1145/357062.357071

[32] LLVM Project. 2022. LLVM Block Frequency Terminology. https://llvm.org/docs/BlockFrequencyTerminology.html.

[33] 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 *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 389–400. https://doi.org/10.1145/2635868.2635900

[34] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 309–329. https://doi.org/10.1007/978-3-030-22038-9_15

[35] microsoft. 2021. New Security Signals study shows firmware attacks on the rise. https://www.microsoft.com/security/blog/2021/03/30/new-security-signals-study-shows-firmware-attacks-on-the-rise-heres-how-microsoft-is-working-to-help-eliminate-this-entire-class-of-threats/.

[36] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. {BinSim}: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *26th USENIX Security Symposium (USENIX Security 17)*. 253–270.

[37] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 757–768. https://doi.org/10.1145/2810103.2813617

[38] MNEMONIC LABS. 2020. Uncovering vulnerabilities in pacemakers. https://www.mnemonic.no/blog/uncovering-vulnerabilities-in-pacemakers/.

[39] Carey Nachenberg. 1997. Computer virus-antivirus coevolution. *Commun. ACM* 40, 1 (1997), 46–51. https://doi.org/10.1145/242857.242869

[40] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*. 177–189. https://doi.org/10.1145/3359789.3359812

[41] Oreans Technologies. 2022. Themida Overview. https://www.oreans.com/themida.php.

[42] Anthony Ralston, Edwin D Reilly, and David Hemmendinger. 2000. *Encyclopedia of computer science*. Grove's Dictionaries Inc. 1514–1515 pages.

[43] 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 *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 142–157. https://doi.org/10.1145/3453483.3454035

[44] Kevin A Roundy and Barton P Miller. 2013. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)* 46, 1 (2013), 1–32. https://doi.org/10.1145/2522968.2522972

[45] 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? *ACM Computing Surveys (CSUR)* 49, 1 (2016), 1–37. https://doi.org/10.1145/2886012

[46] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 94–109. https://doi.org/10.1109/SP.2009.27

[47] statista. 2020. Number of Connected IoT Devices Worldwide. https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/.

[48] Mechthild Stoer and Frank Wagner. 1997. A simple min-cut algorithm. *Journal of the ACM (JACM)* 44, 4 (1997), 585–591. https://doi.org/10.1145/263867.263872

[49] Tencent Blade Team. 2021. Exploiting Qualcomm WLAN and Modem Over The Air. https://blade.tencent.com/en/advisories/qualpwn/.

[50] Roberto Tiella and Mariano Ceccato. 2017. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 182–192. https://doi.org/10.1109/SANER.2017.7884620

[51] 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 *2015 IEEE Symposium on Security and Privacy*. IEEE, 659–673. https://doi.org/10.1109/SP.2015.46

[52] Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Enhancing DNN-Based Binary Code Function Search With Low-Cost Equivalence Checking. *IEEE Transactions on Software Engineering* (2022). https://doi.org/10.1109/TSE.2022.3149240

[53] 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. *arXiv preprint arXiv:2205.12713* (2022). https://doi.org/10.48550/arXiv.2205.12713

[54] 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

[55] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330. https://doi.org/10.1109/ASE.2017.8115645

[56] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237. https://doi.org/10.1145/2970276.2970312

[57] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. 2018. Manufacturing resilient bi-opaque predicates against symbolic execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 666–677. https://doi.org/10.1109/DSN.2018.00073

[58] 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 873–884. https://doi.org/10.1109/ICSE43902.2021.00084

[59] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–387. https://doi.org/10.1145/3395363.3397361

[60] 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

[61] Peihua Zhang. 2022. *Khaos*. https://doi.org/10.5281/zenodo.7496594

[62] zynamics GmbH and Google LLC. 2022. BinDiff Manual. http://www.zynamics.com/bindiff/manual/index.html.

Received 2022-09-02; accepted 2022-11-07