Khaos: The Impact of Inter-procedural Code Obfuscation on Binary Differencing 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‡ labiyuanming@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 differencing 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 differencing 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 fis-sion 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 differencing techniques (less than 19%) with lower runtime overhead (less than 7%).

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

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 differencing 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 differencing 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 differing 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 differencing 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 differencing 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 differencing 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 differencing 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 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.

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 obfuscation 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., \( x^2 \neq -1 \)) 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-procedural 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_f11e(). 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 ([2][3]) 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 ([6][7][8]) to create the call relationship of two sepFuncs. Basic block ([8]) 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 ([6]) 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 oriFuncs 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 sepFuncs, and finally rebuilds the control flow and the data flow among sepFuncs 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 sepFuncs and remFunc to ensure correctness. If the fission generates too many sepFuncs, 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 sepFuncs 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.
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 sepFuncs, we also need to identify the inputs and the outputs of these regions to construct the parameters and return value of sepFuncs. 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
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 set jmp/long jmp and the C++ exception handling (EH in short). The fission requires special handles of them.

Handling the set jmp/long jmp. Programmers could use the set jmp() in a function to record the current context into a jmpbuf structure. And then, they could use the long jmp() 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 set jmp(). There is a requirement here that the set jmp() and the long jmp() using the same jmpbuf must be in the same call chain. Therefore, the call-site of the set jmp() cannot be separated into any sepFunc, because the stack frame of the function that calling the set jmp() cannot be freed when the corresponding long jmp() is executed. Otherwise, the long jmp() 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 fusFuncs. 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
We choose the 2nd bit and the 3rd bit of function pointers (a) Before fusion.

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

we compress them into one 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 oriFuncs need to be removed, and all call-sites to the oriFuncs 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 oriFuncs. 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 oriFuncs, 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 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
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 \( \text{Current} \), and writes two local variables \( \text{tmp1} \) and \( \text{tmp2} \) 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 \( \text{fusFunc} \) at the same time, adding data dependency and control dependency so that the \( \text{fusFunc} \) 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 \( \text{sepFuncs} \) generated by the fission. In this case, the issue of handling indirect function calls no longer exists;
- **FuFi.ori**: Only aggregating the \( \text{oriFuncs} \) 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 \( \text{sepFuncs} \) and the fission-unprocessed \( \text{oriFuncs} \) 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 \textsc{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 \textsc{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 \textsc{Khaos} in terms of effectiveness and performance, and answers the following questions:

- **(Q1)** How is the performance of the obfuscated programs?
- **(Q2)** How does \textsc{Khaos} work against the state-of-the-art binary difﬁng techniques?
- **(Q3)** How good is \textsc{Khaos} at hiding real vulnerable code?

**Test Suites.** We used three test suites to evaluate \textsc{Khaos}: 1) All C/C++ programs in SPEC CPU 2006/2017 benchmarks with the \textit{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...
We separately evaluated the performance overhead of the fission and the fusion, and the three combination modes introduced in §4.1. 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.

Comparison targets. To compare with existing obfuscators, 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 differencing 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 differencing 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).

4.2 The Effectiveness Against Binary Differencing

Comparing binary differencing 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 differencing tool BinDiff [62], we normalized its similarity score to \([0, 1]\). For other tools open-sourced 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).

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 sepFuncs 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 .nd_file). 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].

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

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 cannot 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.ori, 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

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>CVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>JerryScript</td>
<td>opfunc_spread_arguments</td>
<td>2020-13991</td>
</tr>
<tr>
<td>QuickJS</td>
<td>compute_stack_size_rec</td>
<td>2020-22876</td>
</tr>
<tr>
<td>BusyBox1.33.1</td>
<td>getvar_s</td>
<td>2021-24282</td>
</tr>
<tr>
<td></td>
<td>handle_special</td>
<td>2021-24284</td>
</tr>
<tr>
<td>OpenSSL 1.1.1</td>
<td>init_sig_algs</td>
<td>2021-3449</td>
</tr>
<tr>
<td></td>
<td>EC_GROUP_set_generator</td>
<td>2019-1547</td>
</tr>
<tr>
<td>libcurl 7.34.0</td>
<td>suboption</td>
<td>2021-22925</td>
</tr>
<tr>
<td></td>
<td>init_wc_data</td>
<td>2020-8285</td>
</tr>
<tr>
<td></td>
<td>conn_is_conn</td>
<td>2020-8231</td>
</tr>
<tr>
<td></td>
<td>ftp_connect</td>
<td>2019-5482,2019-5436</td>
</tr>
<tr>
<td></td>
<td>ftp_state_list</td>
<td>2018-1000120</td>
</tr>
<tr>
<td></td>
<td>alloc_addbyter</td>
<td>2016-8618</td>
</tr>
<tr>
<td></td>
<td>Curl_cookie_getlist</td>
<td>2016-8623</td>
</tr>
<tr>
<td></td>
<td>ConnectionExists</td>
<td>2014-0138</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2015-3413</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19</td>
</tr>
</tbody>
</table>

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 the 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/zzenodo.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.

A.3.3 BinTuner Part. The diffing files of BinDiff for the binary files generated by BinTuner is in the diffing/BinDiff/ 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 ??.

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.

References


