DAMGate: Dynamic Adaptive Multi-feature Gating in Program Binaries

Yurong Chen, Tian Lan, Guru Venkataramani
{gabrielchen,tlan,guruv}@gwu.edu

ABSTRACT

Feature creep has emerged as a serious threat due to the growing number of utilities and capabilities crammed into modern software systems. While feature elimination and de-bloating techniques can produce slimmer executables, a complete elimination of all unnecessary or unwanted features is often not possible, not only due to the tight coupling of feature-related functions/codes, but also because the usefulness/necessity of program features is often difficult to determine statically and can vary during runtime. This paper presents DamGate , a framework for dynamic feature customization, allowing vigilant management of program features at runtime to prevent violation of privacy and security policies. At the heart of this technique is the selective placement of checker functions (known as *gates*) into feature-constituent functions that need to be protected. Through execution gating and feature validation on the fly, DamGate provides differentiated control policy for program features and enables flexible runtime reconfiguration. The proposed framework is prototyped and evaluated using LibreOffice, a large-scale office suit. The evaluation results show that it can achieve desired feature customization with negligible gating overhead.

KEYWORDS

feature customization; de-bloating; binary rewriting

1 INTRODUCTION

Modern software systems are typically crammed with diverse capabilities and utilities (known as program features) to facilitate code reuse and enable compatibility under different deployment environments. However, the continuing expansion of software features leads to the problem of feature creep [13], which not only causes growing software complexity, larger installation footprint and runtime overhead, but also results in an increased attack surface with higher possibility of exploitable vulnerabilities, especially in large-scale, object-oriented applications.

To mitigate feature creep, existing approaches often focus on the elimination of undesired or unnecessary software features. Several proposals have been made towards feature separation [22], reduction [13] and code de-bloat [12, 37], such as static analysis and program slicing to trim features either in program source code or through runtime de-bloat [35, 37]. While feature elimination can produce slimmer executables prior to deployment, the permitted features still need to be vigilantly managed at runtime, due to a number of reasons. (i) The usefulness of program features is often difficult to determine statically. It is shown that 83% of available browser features are executed on less than 1% of the most popular 10,000 websites [30]. Although these features are required to guarantee usability, they constitute a significant source of feature bloat, which can only be mitigated via a dynamic management approach. (ii) Permitted features can still lead to serious security issues if they are not managed properly. For instance, system logging and money transfer (both necessary, permitted features) executed simultaneously in a banking system can potentially result in information leakage or even unauthorized behaviors [13]. (iii) Due to entanglement and dependency between features, a complete elimination of unnecessary features while maintaining all desired features may be impossible, e.g., the removal of Google Instant feature also disabled some useful keyboard shortcuts such as “ESC” (jumping to text search box) and “Tab” (jumping to the first search result).

In this paper, we propose DamGate (Dynamic Adaptive Multi-feature Gating), a binary customization tool that enables dynamic management of software features in an adjustable and tailored fashion. After a de-bloated executable is obtained through feature elimination, the remaining (permitted) features are then customized with respect to users' preferences, system security policies and application contexts. DamGate complements existing feature elimination approaches by protecting the de-bloated binaries through dynamic gating and feature validation during runtime. In particular, access to program features that are unnecessary under current security policy or execution context are prohibited by gating, in order to prevent any undesired interaction between admitted features. Our approach enables fine-grained management of program features that are tightly coupled or cannot be permanently removed due to the negative impact on usability. DamGate 's gating approach also enables agile feature reconfiguration as user preference or system environment changes. It provides differentiated feature control and security policies through gates that are customizable on the fly.

A key feature of DamGate is that it performs feature customization on binaries. This is motivated by the fact that many legacy programs, playing a critical role in government and military systems such as the Strategic Automated Command and Control System used in Department of Defense [6], often do not have source code available. With more components and abstractions retrofitted to existing software systems to meet the continuously growing requirement, feature creep has been an obtrusive and hard issue in commercial software and legacy systems [6, 10, 13]. Our DamGate leverages existing binary rewriting tools such as [26, 29, 31] to analyze and instrument binaries, in order to enable feature customization for legacy programs.

DamGate consists of two main modules, namely feature identification and feature customization. First, after disassembling and conducting control flow analysis of binaries, DamGate identifies

FEAST'17, November 3rd, 2017, Dallas, TX, USA.
© 2017 ACM. ISBN 978-1-4503-5395-3/17/11…$15.00
DOI: http://dx.doi.org/10.1145/XXXXXX.XXXXXX

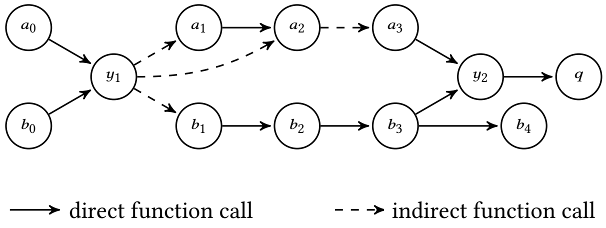


Figure 1: Running example: call graph of printer system

and extracts target features that are defined through specific seed functions from program call graphs. Examples of seed functions include critical functions that are related to user privacy and program capability, and the core functions that enable certain services required by implementation of program features. Next, to protect each feature from unauthorized access, DamGate places "gates" (which are checker functions) to filter function calls from or to other features/functions. While direct function calls have fixed callees, the addresses of indirect callees are undetermined prior to runtime, which requires further analysis. Thus, we develop two different mechanisms for placing gates at direct and indirect function calls/jumps, respectively. The binaries are then instrumented in a way that administrators can conveniently modify the gates and update gating policies, thus enabling quick reconfiguration and management of diverse program features.

We implement a prototype of DamGate . To fully automate feature identification and customization, we leverage a number of tools from binary static analysis (CodeSurfer [2]), dynamic analysis (Pin [26]) and rewriting (Dyninst [31]). The output of DamGate is a de-bloated, modified version of binary executables with feature customization enforced through gating. We evaluate the effectiveness of DamGate on real-world applications such as LibreOffice, an open source office software suite. We show that DamGate succeeds in identifying and customizing various features, and in preventing the unwanted interactions among different features. The number of instructions for placing each direct and indirect gate is around 70 and 150, respectively. The total instruction increase of DamGate is around 0.0068% compared with original programs.

In summary, this paper makes the following contributions:

- We design DamGate , an automated tool that enables feature identification and customization with binaries. It provides dynamic management and protection of different program features.
- By placing gates at direct and indirect function calls/jumps, DamGate enables dynamic feature reconfiguration to be adaptive to changing security policies and user preferences.
- DamGate is evaluated on real-world, large-scale applications such as LibreOffice. The results show that DamGate can achieve the desired feature customization with negligible gating overhead.

2 MOTIVATION

Our DamGate framework for dynamic feature customization is motivated by the fact that simply admitting all necessary program features (after elimination) can still pose serious security risks. More precisely, since program features are often tightly coupled, without proper protection and isolation of feature-constituent functions, any undesired interaction of different features (e.g., evoking functions belonging to a logging feature when executing other banking features) can give rise to security threats such as information leakage and privilege escalation [15]. Dynamic customization also becomes vital in security systems that require multi-level security management, as it enables different access to program features to be set up for different groups of users, e.g., in networked printers with different WAN-related features that can be exploited for DoS or even physical attacks [21].

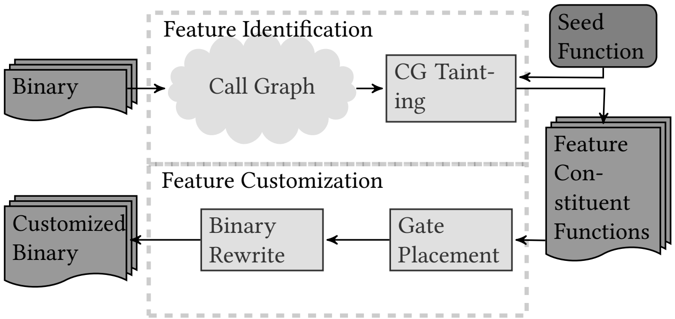


Figure 2: DamGate System Diagram

Dynamic feature customization is especially hard for legacy software whose source codes are usually unavailable. Hence DamGate focus on binaries. Due to the lack of debugging information in stripped binaries, program functionality is normally difficult to interpret regarding code semantics and control flows, let alone to analyze and instrument the program for features customization. Furthermore, the optimizations carried out during multiple phases such as compiling and linking can divide function bodies apart, making it even harder to acquire the necessary information related to features.

In this paper, we focus on feature customization and assume that improper control flow transfers such as ROP and JOP can be protected by existing techniques such as ASLR (Address Space Layout Randomization) [42] and CFI (Control Flow Integrity) [1, 43]. We also do not consider self-modifying codes which can be analyzed by other techniques [4, 27].

Figure 1 shows an illustrative example of two coupled features from a printer system, where a, b and y denote functions related to networking, printing and logging, respectively. These three features are permitted after feature elimination.

When function y_1 is called (by a_0 or b_0), it will log some information of the caller (e.g., current system states) and then pass it to its callee function (a_1 or b_1). When networking and printing features are permitted at the same time, y_1 can possibly take the information from a_0 (or b_0) and transfer it to b_1 (or a_1). Moreover, when b_3 is supposed to jump to its own feature function b_4, it can be redirected to y_2, which results in undesired logging. Both of these two situations can lead to risky states that affect security and privacy of the printer system. A more detailed analysis of this example will be provided in section 3.

3 SYSTEM DESIGN

The system diagram of DamGate is depicted in figure 2. The goal of DamGate is to customize the execution of remaining feature functions after eliminating unwanted program features with de-bloat. It takes a binary (executable or shared object files) and a set of seed functions (for feature identification) as inputs. The binary will be disassembled into assembly code to perform static call graph analysis. In parallel, we will run the binaries and analyze its dynamic call graph. By combining both static and dynamic call graphs, a relatively precise call graph (CG) is generated for feature identification. Relying on the input seed functions that define unique feature operations, capabilities, or system service access, we identify all constituent functions for each feature on the CG. Next, we develop an algorithm for choosing functions that need to be gated along various execution paths to enable feature customization, after which binary rewriting tools are used to insert gates into binaries. The gates will separate different features and protect the target feature from being infiltrated by other features.

We notice that direct and indirect function calls require different treatment in both call graph generation and gating. While the callee of direct function calls can be easily identified in assembly, the indirect function calls can not be statically decided. Thus, dynamic call graph methods are employed to obtain the exact call sequences of target features. These two types of function calls also need different gating mechanisms. Our DamGate consists of two major modules, feature identification and gate placement, which are detailed in the rest of this section.

3.1 Feature Identification

For feature identification, we assume that the information of **seed functions** are available to us to bootstrap and identify various program features. Formally, we define program features as follows.

Definition 3.1. **Program Feature:** Each feature, denoted by F^i, is defined by a set of constituent functions, e.g., $F^i = \{f_1^i, f_2^i, ..., f_n^i\}$. Further, for each feature i, there exists a seed function $f_s^i \in F^i$, which uniquely represents the type of operation, utility, or capability of the program feature[1]. All other constituent functions in the set F^i are located on the execution paths leading to seed function f_s^i. The set of all program features is denoted by $\mathcal{F} = \{F^1, F^2, ..., F^m\}$.

Definition 3.2. **Admitted Program Feature:** A subset of program features, $\mathcal{A} \subset \mathcal{F}$, that are allowed to be executed in the current environment or under current user's privilege, is defined as the admitted program features $\mathcal{A}$.

For the example depicted in figure 1, seed function a_3 and b_3 are given, and the call graph related to them is constructed. Suppose that the features associated with seed a_3 and b_3 are F^1 and F^2, respectively. Then $F^1 = \{a_0, y_1, a_1, a_2, a_3, y_2, q\}$, $F^2 = \{b_0, y_1, b_1, b_2, b_3, y_2, b_4\}$. If the admitted feature is F^1, then functions that belong to F^2 should not be accessed, thus requiring gates to be placed along the execution paths.

We adopt call graph analyses to identify each feature F^i, by extracting the functions along the execution paths that lead to each target seed function. Static analysis alone cannot be sufficient since they will not resolve the indirect control flow transfers. At the same time, dynamic call graph only represents one specific run of the program. On the other hand, symbolic execution [29, 41] can provide a high code coverage but are not feasible in practice due to the problem of path explosion in large-scale software systems.

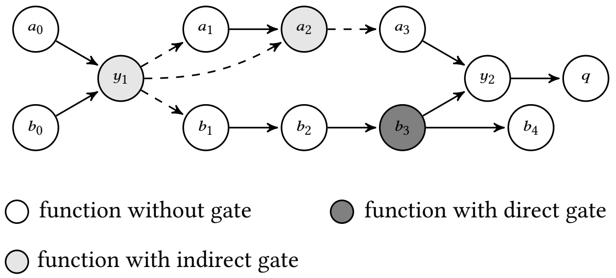


Figure 3: Illustration of Gating using a running example

To this end, we combine the information from static and dynamic call graph analysis for better feature identification. Once static call graph is generated, we annotate the function calls with their call types, i.e., direct or indirect call. Then we perform dynamic instrumentation to explore more function calls as follows: (1) We execute the program and guide the execution to each leaf node in the static call graph. If there are still control flow branches inside the leaf node, every branch will be marked and executed using the forced execution technique in [38]; (2) For caller functions annotated with indirect call, we force the program to go through all branches in this caller function and to expand the call graph.

We collect the information from both static and dynamic analyses to build a combined call graph as the base of feature identification. To facilitate further analysis, direct function calls that belong to the same features are merged as a supernode in the call graph. No feature checking needs to be performed inside a supernode because the function calls cannot lead to other features.

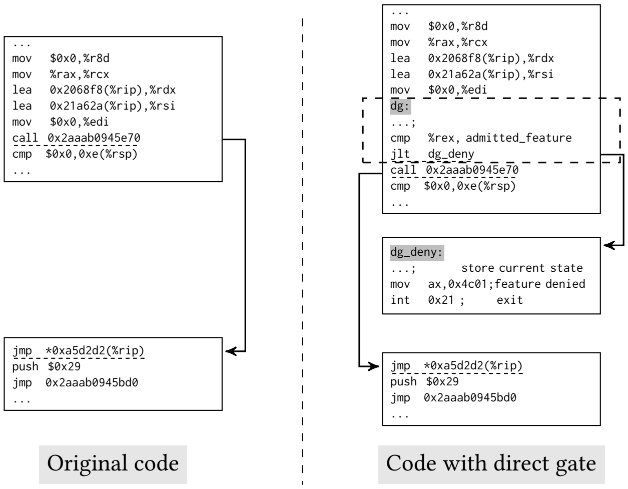
After the execution paths associated with the target feature i are extracted, all the functions belonging to these paths will be tainted as elements of the set F^i. We iteratively perform this extraction and tainting procedure for all features of interest in the program binaries.

3.2 Feature Customization

For feature customization, we insert *gates* in different feature-constituent functions to prevent the execution of unpermitted features/functions. Without loss of generality, consider a single admitted program feature F^i, while other features F^j for $j \neq i$ are not allowed. Our DamGate places **Gate** selectively in functions from set F^i to ensure that the control flow transfers always happen within the perimeter of feature F^i. If function calls go beyond the admitted feature, the gate will throw an exception and terminate the execution.

Definition 3.3. **Gate** A gate is a checker function that is inserted by DamGate into the original binary code. Gates ensure that the current execution stays in constituent functions belonging to admitted program features $\mathcal{A}$ and terminates the execution if it steps beyond the permitted boundary.

[1] Complex program feature may contain more than one seed functions. Our system model in this paper can be easily extended to take multiple seed functions into account



```
...
mov   $0x0,%r8d
mov   %rax,%rcx
lea   0x2068f8(%rip),%rdx
lea   0x21a62a(%rip),%rsi
mov   $0x0,%edi
call  0x2aaab0945e70
cmp  $0x0,0xe(%rsp)
...
```

```
jmp  *0xa5d2d2(%rip)
push  $0x29
jmp  0x2aaab0945bd0
...
```

```
...
mov   $0x0,%r8d
mov   %rax,%rcx
lea   0x2068f8(%rip),%rdx
lea   0x21a62a(%rip),%rsi
mov   $0x0,%edi
dg:
...;
cmp   %rex, admitted_feature
jlt   dg_deny
call  0x2aaab0945e70
cmp  $0x0,0xe(%rsp)
...
```

```
dg_deny:
...;        store current state
mov   ax,0x4c01;feature denied
int   0x21 ;     exit
```

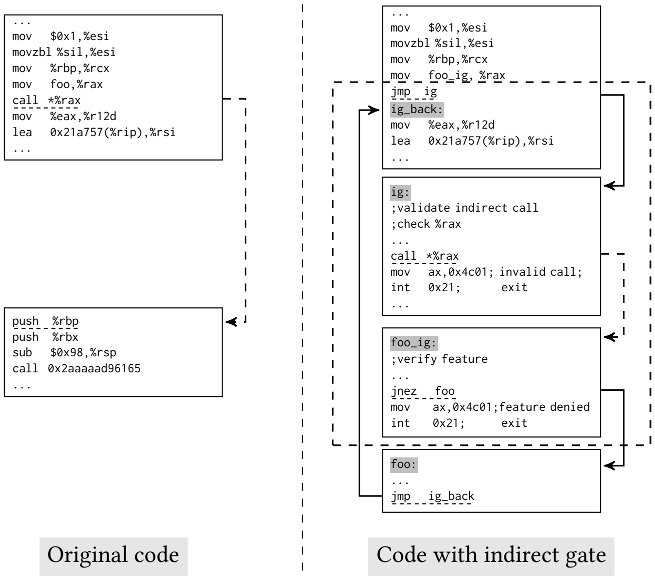
```
jmp  *0xa5d2d2(%rip)
push  $0x29
jmp  0x2aaab0945bd0
...
```

Figure 4: Binary rewriting for direct gating. Gate-related instructions are in the dashed box.

In an aggressive gating scheme, every functions that are tainted with admitted features will be gated, and this will likely incur prohibitive runtime overhead. Based on the assumptions in section 2, we propose the following light-weight gating strategy on given control flow graphs: (1) For direct function calls, we place gates to check whether the features are permitted to determine the legitimacy of current execution. The control flow transfer of direct function calls are not checked as mentioned in section 2; (2) Function returns are not instrumented, assuming that control flow integrity is protected using techniques mentioned in section 2; (3) Along the execution path of each permitted feature, if the forking nodes in the call graph are gated, then merging nodes are considered as safe because only branches with the admitted feature can lead to them. Thus, we do not place gates at merging nodes (while we note that this only applies to direct function calls).

For indirect function calls along the execution path, we check both the legitimacy of control flow transfers and the associated features of the callee function.

As mentioned previously, the differences between direct and indirect function calls require different gating mechanisms. We use the term **direct gates** for the codes checking direct function calls and **indirect gates** for the others. Let *DG* and *IG* denote the sets of direct and indirect gates, respectively. In summary, the target functions, in which direct and indirect gates are placed, are selected through the following steps: (1) For direct function calls, a function is gated only if it satisfies two conditions. (a). It is a forking node in the call graph, e.g., multiple functions can be invoked from this caller; (b). At least one of the callee functions belongs to a different feature other than the features associated with the caller function. It is easy to see that if the callee and caller have exact the same set of features, then no gate needs to be placed to differentiate the executions. Thus, the caller function of direct calls are assigned to set *DG*; (2) All indirect function calls are assigned to *IG*, and all associated features of each callee function will be checked against the set of admitted program features $\mathcal{A}$, to determine the legitimacy of an execution. The design of direct and indirect are detailed next.



```
...
mov   $0x1,%esi
movzbl %sil,%esi
mov   %rbp,%rcx
mov   foo,%rax
call  *%rax
mov   %eax,%r12d
lea   0x21a757(%rip),%rsi
...
```

```
push  %rbp
push  %rbx
sub   $0x98,%rsp
call  0x2aaaaad96165
...
```

```
...
mov   $0x1,%esi
movzbl %sil,%esi
mov   %rbp,%rcx
mov   foo_ig, %rax
jmp   ig
ig_back:
mov   %eax,%r12d
lea   0x21a757(%rip),%rsi
...
```

```
ig:
;validate indirect call
;check %rax
...
call  *%rax
mov   ax,0x4c01; invalid call;
int   0x21;      exit
...
```

```
foo_ig:
;verify feature
...
jnez   foo
mov    ax,0x4c01;feature denied
int    0x21;      exit
```

```
foo:
...
jmp   ig_back
```

Figure 5: Binary rewriting for indirect gating. Gate-related instructions are in the dashed box.

3.3 Direct Gates

We add a direct gate before each direct function call for the functions in set *DG*. Before runtime, functions of interests will be tainted with appropriate features through the call graph analysis. The admitted program features (denoted as feature index) are hard-coded into each function to reduce runtime overhead. Caller functions will store the feature index of callee functions. On the other hand, the index of admitted features are stored in the configuration file to enable dynamic management and reconfiguration under different security policies and environments.

A direct gate will retrieve the list of associated features of both caller and callee functions, then compare them with the set of admitted program features. Only when the caller and callee functions both belong to the admitted program feature $\mathcal{A}$, shall the execution proceed to the callee function.

As illustrated in figure 4, a direct gate labeled as *dg* is placed before the original function call. If the current feature access is denied, the execution will be redirected to *dg_deny*, which is a system interrupt. Otherwise, the result of *dg* verification is validated, meaning that the features of both original caller and callee functions are within the set of admitted program features. The direct gate will guide the program to the original call site and continue its normal execution.

3.4 Indirect Gates

In order to gate indirect function calls for feature customization, we first need to identify all indirect call sites, denoted by the set *IG*. This is achieved through the following steps: (1) From the dynamic call graph generated in section 3.1, we perform cross inference to recursively resolve part of the indirect function calls. This will reveal most of the relevant indirect calls. (2) For indirect function calls whose calling addresses are hard-coded, we can easily find indirect code entries from relocation table. (3) In addition to the two steps above, we also over-approximate the possible indirect callee functions using VSA(Value Set Analysis) as mentioned in [3].

The indirect function calls identified through the above steps will be considered as valid control flow transfers and DamGate creates a trampoline function for each valid indirect function calls in the protected memory region. The protected memory region has a special address format [42]. DamGate enforces the CFI (Control Flow Integrity) of these valid indirect calls by implementing the approach mentioned in [16, 42]. When indirect function call occurs, it is redirected to the associated trampoline function. Different from the checkings in [42], indirect gates in DamGate will check both the control flow integrity and feature legitimacy of the target function calls. Before the trampoline function is called, the indirect gate will check if the callee address resides in the protected memory region. Additionally, before the trampoline function jumps back to the original callee function, another check will be performed to verify if the feature of callee function is permitted.

The control flow transfers of indirect gating are shown in figure 5. Suppose the original indirect function call happens when the address of function foo is loaded into register $\%rax$ and $call$ $^*\%rax$ is executed. Instead of letting foo get invoked, the indirect gate will replace the address of foo with a prefixed trampoline function foo_ig. Before calling foo_ig, we'll check the format of foo_ig's address to make sure that it's the function in the protected memory region. Once validated, function foo_ig will be invoked. Function foo_ig will perform feature verification then jump to the actual implementation of function foo if current function feature is admitted. At the end of function foo, a jump instruction will lead the program back to the next instruction of original foo call site, labeled as ig_back.

We illustrate the policy of gate placement in figure 3 using the example from figure 1. Function y_1 can invoke a_1, b_1 and a_2 through function pointers as denoted by dashed line between functions.Function a_2 is also the caller of an indirect call. As such, y_1 and a_2 will be checked by direct gates. Function b_3 is a forking node and only involves direct function calls, so it is checked by a direct gate.

4 IMPLEMENTATION

This section shows the tools we use to achieve the design of section 3. DamGate relies on several binary analysis and instrumentation tools to achieve our goal of identifying features, protecting function calls and rewriting binaries.

We utilize both static and dynamic analyses to get a more accurate and representative call graph from binaries and object files. CodeSurfer [2], a static analysis tool, is used for static binary call graph generation. It can investigate properties and behaviors of binaries, including CFG generation. CodeSurfer incorporates IDAPro to parse the input binary file and generate an initial version of CFG, followed by VSA (Value-Set Analysis) and ASI (Aggregate Structure Identification) to further analyze indirect jumps and calls. However, the discover of indirect calls are still not precise in CodeSurfer and the dynamic binary instrumentation tool Pin is used to generate the dynamic call graph (execution path) of specific runs [11, 26]. By combing static and dynamic call graph, we can recursively explore the possible indirect function calls and complete the control flows for features.

	Direct Gate		Indirect Gate	
Feature	# gates	avg. # instruction	# gates	avg. # instruction
Save file	13	75	106	150
Insert Image	22	67	91	150
Print file	16	70	64	153

Table 1: Gating statistics on LibreOffice

In the feature customization module, we use Dyninst [31] to statically rewrite the binaries for both direct and indirect gates using different policies as described in section 3. Dyninst provides APIs for instrumenting binaries with which we create a "mutator" program to perform the modification on "mutatee" program (the original binary). The resulting binary will have the gating policies enforced. As mentioned in section 3.3, a separate configuration file is also created to store the information of currently admitted features.

5 EVALUATION

We perform and evaluate DamGate on LibreOffice binary, a cross-platform office software suite. All experiments are conducted on an 8-core 3.4GHz Intel i7-3770 server with 16GB RAM.

5.1 overhead

As shown in table 1, the number of instructions for each direct and indirect gate is around 70 and 150, respectively. Within each feature, we notice that the children of a forking point usually contain all the features of their parent, e.g., the features of children is a superset of feature of parent. In this case, if a direct gate is placed in the forking function, the function calls will always be allowed as long as the caller has the admitted feature. This greatly reduces the number of direct gates placed within a feature. These children functions, such as setting up display parameters, are typically near the leaves of call graph and perform basic tasks that can be reused by multiple features. The separation of features are mainly achieved by indirect gates as indicated by the number of direct and indirect gates in table 1.

5.2 protection

When the customized binary is produced, it's executed multiple times to test its effectiveness of protection. The protection goal is to only allow features that are specified in the configuration file. To enforce different protection policies, no modification is required on the binary but on the configuration file. Ideally, features not in the configuration file cannot be executed after gating. This can be easily achieved by gating at more functions. As mentioned in section 3, we selectively place gates to reduce the redundant checks. Hence, if there are multiple entries for a certain feature, it may still be possibly reached when one of the entries is blocked.

We evaluate the level of protection using the number of functions that are unique to the target feature, number of common functions shared with other features and the number of functions unique to other features as shown in figure 6. Note that the common functions for program initializations, which are not directly related to feature

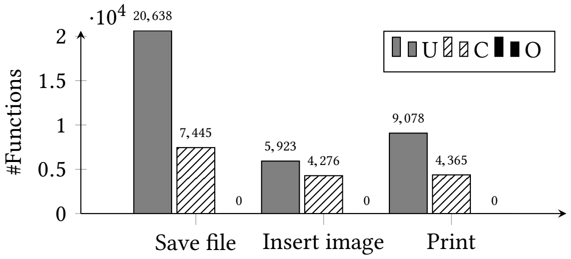


Figure 6: Statistics of functions that can still be accessed after gating: U, C and O stand for unique functions that only belong to current feature, common functions shared by current feature and other features, functions that don't belong to current feature, respectively.

implementations, are excluded from the second bar of each feature.

5.3 case study

LibreOffice is an office software suite with hundreds of features. After the program launches and finishes initialization, it will keep listening and yielding to the next event. Different functionalities are invoked by callback mechanism when user operations are detected. We first put gates along the execution path from main to callback point, then gate functions within each feature.

From our binary analysis, approximately two million functions are contained in LibreOffice binaries and libraries, from which about 193519 are necessary for the three features we evaluated as shown in figure 6 and the rest can be removed by de-bloating techniques. Our customization only incurs a slight code increase of about 0.0068%.

6 RELATED WORK

Efforts have been made towards program feature extraction and decomposition from both research and industrial areas. With increasing number of functionalities being added to existing code base, the software can be notoriously bloated which further leads to security and performance issues.

Bloat Analysis: Plenty of works have been done to analyze and ameliorate both static and dynamic code bloat [20, 37] . Yufei Jiang et al. utilize program slicing methods and data flow analysis to achieve feature removal [13]. Given the function to be removed, they discover and delete codes related to its return value, parameter and call site through the whole program. Jred [12] lifts Java bytecode into Soot IR then removes unused methods discovered from program call graph. After trimming, IR is re-transformed into Java bytecode to produce a light-weight program. While static de-bloating mainly aims at removing unwanted functions to reduce code size, dynamic de-bloating targets at improving runtime performance by detecting inefficient memory usage and redundant instructions. Guoqing Xu et al. propose a profiling approach to summarize the data copies during runtime, rooted from the observation that intensive copies typically indicates excessive program activities. Analyses of copy activities are conducted to find hot copy chains thus providing suggestions for programmers to achieve potential performance gain. Other works by Xu [35, 36] harnesses dynamic slicing to discover low-utility data structures where the cost of generating such data structures are higher than the benefit of using them. Khanh Nguyen et al. [23] design *Cachetor* to pinpoint operations that keep generating identical values. Such values are cached for later use thus to reduce runtime bloat.

While the goal of above works is to de-bloat with respect to code size or runtime performance, we focus on dynamic reconfiguration of features after code de-bloating. Moreover, most of the de-bloating approaches have the limitation of only working with object oriented programming languages while our proposal, DamGate , can be directly applied to binaries. We leverage a series of binary analysis and rewriting tools to achieve this.

Binary Analysis and Rewriting: Binary code analysis is necessary to enable other analyses such as reverse engineering [32], debugging [14] and vulnerability examination [17, 28, 34]. DamGate requires tools that can generate call graph from binaries [2, 7, 11, 18, 38] and perform binary rewriting [26, 31]. FXE [38] is proposed to construct control flow graphs from binaries by forcing the program to execute both branches of each condition in a virtual environment. The address of branches not taken at the first iteration will be saved and executed later. Similar to this idea, we also force the program to explore possible function calls in each branch. However, we will first generate a static call graph using CodeSurfer [2] and later we only perform this enforcement at selected functions based on existing information of static call graph and avoid redundant explorations. Trin-Trin [11], a dynamic call graph generator, utilizes Pin API to track threads and processes then produce per-thread call graphs. By analyzing, merging and pruning these per-thread call graphs, the call graph for the whole program will be created. This approach will also include the system calls. In DamGate , instead of exploring in depth of the call graph of a certain feature, it's preferable that the call graph can grow in width where functions that can fork and invoke multiple targets (especially indirect calls) are captured. The combination of static and dynamic analysis can also be applied to fields such as program vulnerability identification [41], bound check removal [39] and binary differing [19].

Control Flow Integrity: Different methods and evaluations for control flow integrity have been proposed [5, 8, 9, 24, 25, 33, 40, 43]. CCFIR [42] validates each indirect control transfer by creating a function stub inside a *springboard* memory area (with special address format) and redirecting original transfer to this stub. Address checks are performed before redirecting as well as function return to make sure each indirect call at runtime leads to a predetermined valid target. We use the similar idea to verify both legitimacy of indirect function calls and function features.

7 CONCLUSION

In this paper, we present DamGate , a prototype that customizes binary programs to protect feature executions. DamGate places gates (checker functions) into selected feature constituent functions after identifying features from program call graphs. The customized binary will prevent undesirable control transfers among different features and be easily adapted to different protection policies without being modified. Our evaluation results on LibreOffice, a large-scale office software system, show that DamGate can achieve desired protection with minor runtime overhead of around 70 and 150 extra instructions for each direct and indirect gate, respectively. The total percentage of gating instructions introduced by DamGate to LibreOffice is only 0.0068% compared with the original program.

8 ACKNOWLEDGEMENTS

This work was supported by the US Office of Naval Research (ONR) under Award N00014-17-1-2786 and N00014-15-1-2210. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of ONR.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 340–353.

[2] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86:A platform for analyzing x86 executables. In *Compiler Construction.* Springer, 139–139.

[3] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to recognize functions in binary code. USENIX.

[4] Andrew R Bernat, Kevin Roundy, and Barton P Miller. 2011. Efficient, sensitivity resistant binary instrumentation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis.* ACM, 89–99.

[5] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.

[6] Information Technology Management Issues David A. Powner, Director. 2016. Federal Agencies Need to Address Aging Legacy Systems.

[7] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev. ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction.* ACM, 131–141.

[8] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 901–913.

[9] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on.* IEEE, 179–194.

[10] The Standish Group. 2014. Chaos Report.

[11] Rohit Jalan and Arun Kejariwal. 2012. Trin-trin: Who is calling? a pin-based dynamic call graph extraction framework. *International Journal of Parallel Programming* (2012), 1–33.

[12] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, Vol. 1. IEEE, 12–21.

[13] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *High Assurance Systems Engineering (HASE), 2016 IEEE 17th International Symposium on.* IEEE, 122–131.

[14] Christopher Kruegel, William Robertson, and Giovanni Vigna. 2004. Detecting kernel-level rootkits through binary analysis. In *Computer Security Applications Conference, 2004. 20th Annual.* IEEE, 91–100.

[15] Yongbo Li, Fan Yao, Tian Lan, and Guru Venkataramani. 2016. Sarre: semantics-aware rule recommendation and enforcement for event paths on android. *IEEE Transactions on Information Forensics and Security* 11, 12 (2016), 2748–2762.

[16] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture.. In *USENIX Security Symposium.*

[17] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ACM, 24–35.

[18] Jiang Ming and Dinghao Wu. 2016. BinCFP: Efficient Multi-threaded Binary Code Control Flow Profiling. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on.* IEEE, 61–66.

[19] 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).*

[20] Nick Mitchell and Gary Sevitsky. 2007. The causes of bloat, the limits of health. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 245–260.

[21] Jens Müller, Vladislav Mladenov, Juraj Somorovsky, and Jörg Schwenk. 2017. SoK: Exploiting Network Printers. In *Security and Privacy (SP), 2017 IEEE Symposium on.* IEEE, 213–230.

[22] Gail C Murphy, Albert Lai, Robert J Walker, and Martin P Robillard. 2001. Separating features in source code: An exploratory study. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on.* IEEE, 275–284.

[23] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* ACM, 268–278.

[24] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. *ACM SIGPLAN Notices* 49, 6 (2014), 577–587.

[25] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 914–926.

[26] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* ACM, 2–11.

[27] Kevin A Roundy. 2012. *Hybrid analysis and control of malicious code.* Ph.D. Dissertation. The University of Wisconsin-Madison.

[28] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.. In *NDSS.*

[29] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on.* IEEE, 138–157.

[30] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. 2016. Browser feature usage on the modern web. In *Proceedings of the 2016 ACM on Internet Measurement Conference.* ACM, 97–110.

[31] Open Source. 2016. Dyninst: An application program interface (api) for runtime code generation. *Online, http://www.dyninst.org.*

[32] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling.. In *USENIX Security Symposium.* 627–642.

[33] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference.* ACM, 299–308.

[34] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Security and Privacy (SP), 2017 IEEE Symposium on.* IEEE, 921–937.

[35] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. *ACM Sigplan Notices* 45, 6 (2010), 174–186.

[36] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 3 (2014), 23.

[37] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research.* ACM, 421–426.

[38] Liang Xu, Fangqi Sun, and Zhendong Su. 2009. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep* (2009).

[39] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. 2017. SIMBER: Eliminating Redundant Memory Bound Checks via Statistical Inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection.* Springer, 413–426.

[40] Fan Yao, Jie Chen, and Guru Venkataramani. 2013. Jop-alarm: Detecting jump-oriented programming-based anomalies in applications. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on.* IEEE, 467–470.

[41] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. 2017. StatSym: vulnerable path discovery through statistics-guided symbolic execution. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on.* IEEE, 109–120.

[42] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 559–573.

[43] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries.. In *USENIX Security Symposium.* 337–352.