# Binary Software Complexity Reduction: From Artifact- to Feature-Removal

Masoud Ghaffarinia
The University of Texas at Dallas
800 West Campbell Road
Richardson, Texas 75080
masoud.ghaffarinia@utdallas.edu

Kevin W. Hamlen
The University of Texas at Dallas
800 West Campbell Road
Richardson, Texas 75080
hamlen@utdallas.edu

## ABSTRACT

Research progress and challenges related to the automated removal of potentially exploitable, abusable, or unwanted code features from binary software are examined and discussed. While much of the prior literature on this theme has focused on elimination of *artifacts*—software functionalities unanticipated by the code's original developers (e.g., bugs)—an approach for safely and automatically removing features intended by developers but unwanted by individual consumers is proposed and evaluated. Such a capability is potentially valuable in scenarios where security-sensitive organizations wish to employ general-purpose, closed-source, commercial software in specialized computing contexts where some of the software's functionalities are unneeded. In such contexts, unwanted functionalities can constitute an unnecessary security risk, motivating their removal before deployment.

Preliminary experiments using a combination of runtime tracing, machine learning, and control-flow integrity enforcement indicate that automated software feature removal is feasible for simple binary programs without source code. Future work is proposed for scaling these preliminary results to larger, more complex software products.

## CCS Concepts

•**Security and privacy → Software security engineering;** Software reverse engineering;

## Keywords

Control-flow Integrity; Software Fault Isolation; Code-reuse Attacks; Trace Monitors

## 1. INTRODUCTION

Security of software is widely believed to be inversely related to its complexity (cf., [32, 41]). With more software features, larger implementations, and more behavioral variety inevitably come more opportunities for programmer error, malicious code introduction, and unforeseen interactions between components.

Unfortunately, this danger stands at odds with one of the major economic forces in the software market—the need to mass-produce ever more general-purpose software in order to tractibly meet the broadening needs of the expanding base of software consumers worldwide. Software developers understandably seek to create products that appeal to the widest possible clientele, in order to maximize sales and reduce overheads. This has led to commercial software products of increasing complexity, as developers pack more features into each product they release. As a result, software is becoming more and more difficult to reliably secure as software becomes more multi-purpose and more complex.

*Code-reuse attacks* [4, 7, 8, 26, 27] are one example of the inherent security risks that such feature-accumulation can introduce. In a code-reuse attack, a malicious software user amplifies an otherwise low-severity software bug, such as a memory corruption bug, to hijack the victim software and control it to perform arbitrary actions. For example, *return-oriented programming (ROP) attacks* [24] abuse such data corruptions to overwrite the stack with a series of attacker-supplied code addresses. This causes the victim program to execute attacker-specified *gadgets* (i.e., code fragments at the attacker-supplied addresses) when it consults the stack to return to the current method's caller. The potential potency of a ROP attack therefore depends in part on the variety of gadgets that reside in the victim program's executable memory as it runs [11, 13]. The larger the software, the more code gadgets are potentially available to be co-opted by the attacker, and the more malicious behaviors can potentially be triggered.

One way to frustrate such attacks is to forcibly reduce the complexity of the software's control-flow graph to a specified subgraph of edges that were originally intended by the software's creators to be reachable. For example, *control-flow integrity (CFI)* [1, 2, 3, 17, 20, 21, 28, 29, 40] and *software fault isolation (SFI)* [18, 31, 36] defenses instrument programs with extra security guard code that validates the destinations of jump instructions at runtime. Jumps that attempt to traverse a graph edge not permitted by the security policy are blocked, thereby detecting and averting the attack. Another approach is to randomize the code in such a way that unwanted control-flow edges are reachable with arbitrarily low probability (with respect to a given attacker model) [12, 15, 19, 39]. Our prior work has introduced SFI, CFI, and code randomization protections for large, commercial, binary-only (closed source) software in fully automated solutions [19, 33, 34, 35].

However, all these defenses demand as a prerequisite an adequate policy to enforce. Usually this comes in the form of a whitelist of control-flow graph edges that the policy declares to be safe. When program source code is available, such a whitelist can sometimes be derived from the program source code (e.g., [10]). Alternatively, a conservative, heuristic disassembly of the binary code can be used in binary-only settings (e.g., [34, 35, 40]).

Unfortunately, recent attacks, such as *counterfeit object-oriented programming (COOP)* [25], have demonstrated the exceptional difficulty of deriving control-flow policies conservative enough to preclude hijackings. For example, the semantics of object-oriented programming idioms tend to intentionally embody large control-flow graphs that are prone to attack even when all unintended edges are dropped [5, 14, 23, 37, 38]. One prominent source of abusable edges is method inheritance and overriding, which introduces control-flow edges from all child method call sites to all parent object methods of the same name and/or type signature. This is often enough flexibility for attackers to craft counterfeit objects that traverse only these "intended" control-flow edges—but in an order or with arguments unforeseen by the developers—to hijack the program.

Our experiences with such attacks have led us to conclude that in many operational contexts it is desirable to derive and enforce control-flow policies that exclude even some of the developer-intended flows of programs. As an example, consider a command-line file compression/decompression utility deployed in a security-sensitive operational context where only the decompression logic is ever used. In such a context, removing the compression logic (and any vulnerabilities it may contain) from the binary code of the utility could achieve important reductions in the attack surface of the system. Other plausible scenarios include removal of multi-OS compatibility code in contexts where compatibility with only one particular OS is needed, or removal of parser logic for file/media formats never used in a particular computing environment.

In these scenarios, feature *removal* could potentially take either of the following forms:

- *Code Removal*: The binary code that implements the undesired functionality could be physically deleted (e.g., replaced with zeros).
- *Control-flow Removal*: The binary code that implements the undesired functionality could be made provably unreachable—e.g., by instrumenting all computed jump instructions in the program with logic that prohibits that destination.

In this paper we pursue the latter approach, observing that it subsumes the the former as a special case, and has the potential to more substantially reduce the attack surfaces of software products of modular design. To illustrate, consider a hypothetical program with critical functionality $F_1$ and undesired functionality $F_2$, and assume its implementation contains three binary code blocks $c_1$, $c_2$, and $c_3$. Functionality $F_1$ is implemented by executing $c_1; c_2; c_3$ (in that order), whereas functionality $F_2$ is implemented by executing $c_2; c_1; c_3$. A strict code removal approach cannot safely remove any code blocks in this case, since all are needed by the critical functionality, which must be preserved. However, a control-flow removal approach can potentially delete control-flow edges $(c_2, c_1)$ and $(c_1, c_3)$ to make functionality $F_2$ unrealizable. In the special case that some code block has no in-edges after control-flow removal, it can be deleted completely.

While there has been much prior work on detecting and averting code-reuse attacks through trace monitoring, most of these prior efforts crucially rely on detecting anomalous behaviors specific to traces not intended by developers—not on detecting traces intended by developers but unwanted by consumers. For example, kBouncer [22] detects ROP attacks by monitoring branch histories and censoring those containing illegal returns and gadget-chaining characteristics. But these telltale characteristics are not typically exhibited by attacks that misuse developer-intended software functionalities.

New techniques are therefore needed to identify and mitigate such attacks automatically in closed-source software products. More generally, we advocate a model that differs fundamentally from a *programmer-intent software security* model [9], instead favoring a data-driven, *customer-is-always-right* model in which security policies are derived and enforced strictly on the consumer side of the software deployment chain.

In this paper we report on some preliminary efforts to achieve practical feature removal of binary programs without source code, and discuss avenues for future research. The following are contributions:

- We present a method to reduce the size and complexity of binary software by removing functionalities unwanted by code-consumers (but possibly intended by code-producers) without any reliance on source code or debug metadata.
- We present a new kind of control graph approximation and show that it helps to reduce false negatives in the analysis, affording smaller attack surfaces for complexity-reduced software.
- We propose an entropy-based method for reducing the false positive rate of this new control graph without sacrificing its other benefits.

Section 2 first defines and describes our information gain-based control flow graph with some examples. Section 3 presents preliminary experimental results. Section 4 discusses future work, and Section 5 concludes.

## 2. TECHNICAL APPROACH

The example in Section 1 illustrates that control-flow graph (CFG) edge removal generalizes code removal. Our approach generalizes that paradigm yet further to control-flow *history* removal. Continuing the example, suppose that the program under consideration has an additional critical functionality $F_3$ implemented by sequence $c_1; c_1; c_3$. This prevents removal of edge $(c_1, c_3)$ from the CFG, since doing so would break $F_3$. But an enforcement mechanism that can block edge $(c_1, c_3)$ conditional on it being immediately preceded by edge $(c_2, c_1)$ successfully removes $F_2$ without harming $F_3$.

In this paper, we propose to learn such history-based CFGs by applying machine learning to sets of sample execution traces. Since it is usually easier for code-consumers to exhibit all features they wish to preserve (e.g., through a typical software quality testing regimen), rather than discovering and exhibiting those they wish to remove, we adopt a whitelisting approach: Let $T_1$ and $T_2$ be training sets of program execution traces that exhibit only software features that must be preserved, and let $T_3$ be a testing set that includes traces for both wanted and unwanted features.

$T_1$ and $T_2$ are assumed to be noise-free; every trace exhibited during training is a critical one that must be preserved after control-flow removal. However, we assume there may be additional critical traces requiring preservation that do not appear in $T_1$ or $T_2$. The learning algorithm must therefore conservatively generalize $T_1$ and $T_2$ in an effort to retain desired functionalities and curtail undesired ones.

Traces are defined as sequences of control-flow edge traversals. To accommodate modern code randomization protections (e.g., ASLR), which randomize the base addresses of program memory sections at load-time, we encode the source and destination addresses of edges as section-offset pairs rather than absolute addresses. This keeps them invariant across different randomizations.

Control-flow edge histories are defined as finite-length subsequences of these traces. A history-based control-flow policy can therefore be defined as a set of permissible control-flow edge histories. While the logic for enforcing an entire history-based control-flow policy could be large, the logic needed to enforce the policy at any particular jump origin need only consider the subset of the policy whose final edge begins at that jump origin. This distributes and specializes the logic needed to enforce the policy at any given jump site in the program.

History lengths are not fixed in our model. While an upper bound on history lengths is typically established for practical reasons, our approach can consider different history lengths at different jump sites based on an estimate of the benefits (e.g., based on information gain). In our design, we first suppose there is a fixed size (possibly large) for the histories, and then proceed to accommodate different sized histories.

Our approach organizes the training flows into a probability tree. For a given branch origin $o$, we build the tree as follows: For each branch origin $o$ discovered during training, we construct a probability tree of depth $H + 1$, where $H$ is the maximum size of the history. The root is labeled with $o$. The nodes in the first level correspond to different destinations found for the root address during the training. Each node at level $k \geq 2$ of the tree is labeled with the destination of the $k - 1$st edge traversal that occurred immediately before the traversal history defined by the path from the root to the node's parent at level $k$. Each non-root node in the tree is labeled with an address representing the branch target, and each edge is labeled with a count representing the number of times that particular address was the branch target given its ancestors in the tree.

Figure 1 presents a sample tree for the address `0xb45e`. Two different branches with origin `0xb45e` were observed during training, having destinations `0xb5ea` and `0xb8b0`, and occurring 5 and 10 times, respectively. (So address `0xb45e` was a branch origin 15 times in the training execution traces.) For the branch with origin address offset `0xb45e` and destination address offset `0xb5ea`, three different addresses were observed as the immediately following destinations. So for example, in the training set there exists a trace in which there was a jump with destination address offset `0xb48f`, followed by some non-branch instructions and a jump with origin address offset `0xb45e` and destination address offset `0xb5ea`.

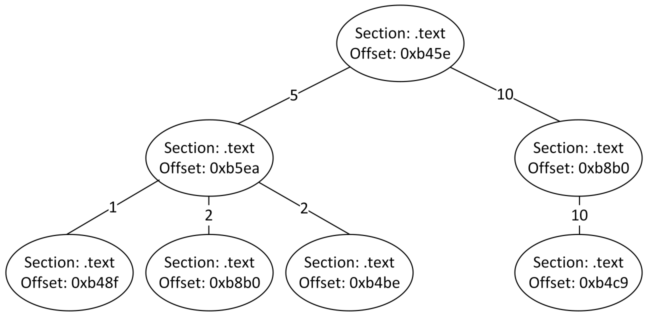

**Figure 1: A probability tree for the branch origin with address offset 0xb45e from the .text section**

It is possible to group the false positive flows into three categories: (1) There could be a valid jump origin that is not in the training traces. (2) There could be a jump with a different destination address not observed during training. (3) There could exist an unobserved history for an observed jump seen during training. Our preliminary experiments suggest that the first case is rare, and can be largely mitigated using binary static analysis to identify likely jump origins. However, the other two cases are common due to the large variety of possible execution traces exhibited by most software.

To address the problems of unobserved (but desired) jump destinations and histories, we must generalize the tree to include more child nodes than were explicitly observed during training. In general, if training observes many diverse jump destinations for a specific subtree, we infer that there is a high chance that there exist additional, desired destinations that were not observed. The same is true for diverse collections of histories. So any node that has many child nodes with low frequencies should be viewed as having low confidence.

To estimate this confidence level, we use entropy to calculate an uncertainty value using the number of times different child nodes of a node appeared in the training. The more diverse the children of a node, the more entropy and uncertainty it has. Using this metric, we remove sub-trees with high entropy to relax the policy and admit more jump destinations and more control-flow histories when jumping from that origin. For example, in Figure 1 we might observe that the entropy of the node with address offset `0xb5ea` is 1.52Sh. If our confidence threshold is less than that, then history constraints are not enforced for jumps from that origin.

## 3. EXPERIMENTAL RESULTS

The test setup can be separated in two sequential parts. The first part is the data generation that produces the execution traces. For this we used the Pin dynamic instrumentation tool [16] to track all branches during each run of each test program. Pin is appealing since it can track multithreaded programs as well as single threaded ones, though we did not use any multithread programs in the preliminary experiments reported here. Data generation requires a test suite based on wanted and unwanted features. The second part of the experiments is the data mining, which consists of a learner that builds the probability trees and prunes certain sub-trees based on the threshold entropy, as explained in Section 2.

| | | New Branch Origins | | New Branch Destinations | |
|---|---|---|---|---|---|
| Command | Samples | Branch Origins | Traces | Branch Origins | Traces |
| `cat -n` | 10 | 0.00 | 0.00 | 4.12 | 13.00 |
| | 100 | 0.00 | 0.00 | 0.00 | 0.00 |
| `chmod -c` | 10 | 0.00 | 0.00 | 0.24 | 1.00 |
| | 100 | 0.00 | 0.00 | 0.73 | 0.30 |
| `dd -ascii` | 10 | 0.00 | 0.00 | 0.71 | 3.00 |
| | 100 | 0.00 | 0.00 | 0.00 | 0.00 |
| `ls -g` | 10 | 1.11 | 3.00 | 3.16 | 17.00 |
| | 100 | 0.47 | 0.60 | 0.63 | 0.40 |
| `ls -l` | 10 | 1.84 | 7.00 | 3.37 | 25.00 |
| | 100 | 0.76 | 0.50 | 0.61 | 0.40 |
| `tar` | 10 | 0.35 | 1.00 | 0.96 | 3.00 |
| | 100 | 0.00 | 0.00 | 0.17 | 0.30 |

**Table 1: Percentage of new flow origins and destinations in test**

An interesting consideration is whether to include unconditional jumps within histories. While one may think that unconditional branches would be useless (since their destinations are fixed), our experiments reveal that including them in the history improves results because it aids in call-return matching. Some calls (e.g., tail-calls) are implemented as jump instructions, but all returns are indirect. Thus, including one but not the other makes it more difficult to detect control-flow anomalies that hijack a return to redirect it to a non-caller.

Another consideration is whether to exclude library control-flows from the program flow since they are shared, and it may be infeasible to learn appropriate policies for them based on profiling only one application that loads them. On the other hand, if security is a priority, the user may be interested in generating a specialized, non-shared version of the shared library specifically for use by each security-sensitive application. For this work, we only consider the branch that targets the shared library and we omit from consideration all subsequent flows until it returns to the program code.

As mentioned earlier in Section 2, we encode addresses as pairs consisting of the section name and the address offset. In this way, ASLR randomization does not affect the results. Also, it avoids side-effects associated with shared library rebasing.

The false positive rates are shown in Tables 1 and 2. These were computed from the test samples using the same features as the trainings. False positives can be categorized into three different types: new branch origins, new branch targets, and new histories for known branches. Table 1 shows the results regarding new branch origins and targets and anomaly detected traces for different programs. Table 2 shows anomalous histories detected. For any of these anomalies the percentage of branch origins for which that kind of anomaly is found and the percentage of execution traces marked as anomalous are shown. For new history anomalies, the results are shown for different history lengths. Also, for the rate of anomaly detected traces for new history anomalies, two numbers are given: the smaller (leftmost) is the rate of traces because of new history anomalies, and the larger (rightmost) is the ratio of total anomaly traces regarding all types of anomalies. Both are shown, since it was observed that if a trace has some kind of anomaly then there is a higher chance to have another type as well.

The programs we exercised in the experiments are all Linux command line programs. We chose these as a starting point for our experiments because of the ease of creating test suites for them. Experiments were conducted for 10, 100, and 1000 training sample traces, with the same number for testing. The tables are averaged over 10 random partitions of traces into training and testing samples over a pull of 2000 trace samples. None of our experiments exhibited any false positives with more than 1000 samples, indicating that our approach is very promising at reasonable sample sizes.

Another interesting result is the percentage of each kind of branch flow among anomalies. We categorized branch flows into return branches, indirect calls, indirect branches (indirect jumps), and system calls. (We did not consider direct branches because their destinations are constant, and therefore there is no need to enforce control-flow guards for them.) Table 3 shows for each program the percentage of each kind of branch flow among anomalous branch origins. Most of the false positive detected edges are return branches. This indicates that our approach might benefit from specialized techniques specifically for return branches (e.g., [30]).

Tables 1 and 2 use the training tree described in Section 2 without generalizing the tree using the entropy threshold. Table 4 shows the results for the same arrangement of samples, but after pruning subtrees with entropy over 1Sh. As shown in the tables, the number of anomalies are reduced mainly when the number of samples are few. This provides a means to derive reasonable approximate policies when there are not enough samples to make a stronger CFI policy. One may notice that the ratio of total anomaly traces in most cases does not change. The reason is that we pruned the tree just for histories and not for branch targets, since relaxing branch targets allows the attacker to misuse the branch and redirect the program as many times as he wants to different locations, making a loop out of the branch with relaxed target [6].

Furthermore, the false negative rate for any program using any number of samples was zero, suggesting that our approach is strong enough to find other sound but unwanted behaviors of programs. To measure false negative rate, we used test samples that apply different command-line arguments (or possibly no arguments) than those specified as consumer-desired functionalities in the training samples. For example, we tested `ls -l` against `ls -a` and `ls -g`, and vice versa. Similarly, we tested `tar` against `tar -Hposix`, along with other programs and features.

The zero false negative rate is mainly achieved via detection of anomalous targets, because any other program behavior has at least one branch target not found in the training. If we only consider anomalous histories, the false negative rate significantly increases. For example, when trained on `ls -l` and tested on `ls -g` with 10 samples for each of the training and test sets, the false negative rate rises to 93% for history size 1, and 71% for history size 5 if anomalous targets are not considered. As an extreme example, when trained on `cat -n` and tested on `cat` with no switches, the false negative rate becomes 100% for all experiments without the benefits of anomalous target detection. We conjecture that higher program complexity appears to lower false negative rates in general for our technique, since the greater complexity highlights anomalous flows, making them easier to detect.

| History size = | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Command | Samples | Branch Origins | Traces | Branch Origins | Traces | Branch Origins | Traces | Branch Origins | Traces | Branch Origins | Traces |
| `cat -n` | 10 | 0.00 | 0.00/13.00 | 1.18 | 2.00/15.00 | 1.18 | 2.00/15.00 | 5.29 | 12.00/20.00 | 6.47 | 15.00/23.00 |
| | 100 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 1.18 | 0.30/ 0.30 | 1.18 | 0.30/ 0.30 |
| `chmod -c` | 10 | 0.00 | 0.00/ 1.00 | 0.00 | 0.00/ 1.00 | 0.00 | 0.00/ 1.00 | 0.24 | 1.00/ 1.00 | 0.24 | 1.00/ 1.00 |
| | 100 | 0.00 | 0.00/ 0.30 | 0.00 | 0.00/ 0.30 | 0.00 | 0.00/ 0.30 | 0.73 | 0.30/ 0.30 | 1.46 | 0.70/ 0.70 |
| `dd -ascii` | 10 | 0.00 | 0.00/ 3.00 | 0.00 | 0.00/ 3.00 | 0.00 | 0.00/ 3.00 | 1.43 | 3.00/ 3.00 | 2.50 | 10.00/10.00 |
| | 100 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 0.36 | 0.10/ 0.10 |
| `ls -g` | 10 | 0.00 | 0.00/18.00 | 1.27 | 12.00/25.00 | 1.90 | 17.00/25.00 | 2.53 | 17.00/25.00 | 3.01 | 20.00/28.00 |
| | 100 | 0.00 | 0.00/ 1.00 | 0.31 | 0.30/ 1.30 | 0.31 | 0.30/ 1.30 | 0.63 | 0.40/ 1.30 | 1.10 | 1.30/ 2.20 |
| `ls -l` | 10 | 0.92 | 13.00/28.00 | 2.30 | 24.00/36.00 | 2.91 | 30.00/36.00 | 3.99 | 30.00/36.00 | 5.21 | 32.00/37.00 |
| | 100 | 0.00 | 0.00/ 0.70 | 0.91 | 0.60/ 1.30 | 1.37 | 0.60/ 1.30 | 2.43 | 1.20/ 1.70 | 3.03 | 1.70/ 2.10 |
| `tar` | 10 | 0.00 | 0.00/ 3.00 | 0.09 | 1.00/ 3.00 | 0.09 | 1.00/ 3.00 | 0.17 | 1.00/ 3.00 | 0.61 | 8.00/ 8.00 |
| | 100 | 0.00 | 0.00/ 0.30 | 0.00 | 0.00/ 0.30 | 0.09 | 0.20/ 0.30 | 0.09 | 0.20/ 0.30 | 0.17 | 0.30/ 0.30 |

**Table 2: Percentage of flow origins having history anomalies and traces found as anomaly in test - Without pruning the training tree**

| | Branch Types | | |
|---|---|---|---|
| Programs | Returns | Indirect Calls | Indirect Branches |
| `cat -n` | 94.00 | 6.00 | 0.00 |
| `chmod -c` | 91.00 | 2.00 | 7.00 |
| `dd -ascii` | 93.00 | 7.00 | 0.00 |
| `ls -g` | 86.00 | 6.00 | 8.00 |
| `ls -l` | 86.00 | 6.00 | 8.00 |
| `tar` | 88.00 | 9.00 | 3.00 |

**Table 3: Percentage of each branch type among anomaly branch origins**

## 4. DISCUSSION AND FUTURE WORK

The preliminary experiments reported here are restricted to small software applications for which distinct functionalities can be clearly defined—e.g., in the form of differing command-line switches and arguments. Future work should consider more elaborate software applications for which desired and undesired functionalities may be more difficult to clearly distinguish. For example, multithreaded applications in which certain threads perform background tasks unrelated to the particular functionalities exhibited in the training set could introduce noise not present in our data set. Such noise may demand elaborations of our entropy-based technique.

Future work should also explore the effects of practical limitations on training set data for certain common types of applications. For example, user-interactive applications present special challenges for constructing comprehensive training sets, since their inputs tend to be time- and context-sensitive. In this paper, we consider only non-interactive applications.

The approach explored in this work is purely data-driven [9]; policies are derived and enforced purely based on runtime trace-monitoring. We speculate that static binary analyses, such machine learning-based disassembly [35], could further improve our results by priming our trace-based learner with code features difficult to identify during dynamic analysis. An example is the branch origin identification suggested in Section 3. In future work we intend to investigate this possibility.

Our entropy-based approach to quantifying confidence on edge origins is one of many possible metrics that might be investigated. We believe this area can be recognized as a new research direction on how to generalize an empirical control flow graph built on sample traces. Improved results might be obtainable from more sophisticated anomaly detection solutions.

Finally, we have not yet investigated the performance tradeoffs associated with different forms of in-lined enforcement of the policies derived from our approach. Evaluating such enforcement strategies is an important avenue for future investigation.

## 5. CONCLUSION

A data-driven, customer-is-always-right approach to automatically removing developer-intended but consumer-unwanted functionalities from binary software was proposed and evaluated. Preliminary results are promising; we were able to safely and automatically exclude functionalities associated with certain command-line options of linux shell commands without harming desired functionalities, and without inspecting or analyzing source code or performing aggressive binary reverse-engineering.

Our approach innovates a probability tree-based control-flow graph approximation in which confidence levels are quantified using entropy. Such policies are shown to be derivable automatically by code-consumers through trace-monitoring, given a suitably large training set of runs.

We believe the development of such enforcement mechanisms will bridge an important gap in modern day software security solutions by mitigating attacks that misuse software features that were intentionally included by software developers of general-purpose, mass-distributed products, but that are unneeded by consumers of these products in more specialized, mission-critical computing contexts.

## Acknowledgments

The research reported herein was supported in part by NSF awards #1054629 and #1513704, ONR award N00014-14-1-0030, and an NSF I/UCRC grant from Raytheon Corporation. Any opinions expressed are those of the authors and not necessarily of the NSF, ONR, or Raytheon.

| History size = | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Command | Samples | Branch Origins | Traces | Branch Origins | Traces | Branch Origins | Traces | Branch Origins | Traces | Branch Origins | Traces |
| `cat -n` | 10 | 0.00 | 0.00/13.00 | 1.18 | 2.00/15.00 | 1.18 | 2.00/15.00 | 5.29 | 12.00/20.00 | 6.47 | 15.00/23.00 |
| | 100 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 1.18 | 0.30/ 0.30 | 1.18 | 0.30/ 0.30 |
| `chmod -c` | 10 | 0.00 | 0.00/ 1.00 | 0.00 | 0.00/ 1.00 | 0.00 | 0.00/ 1.00 | 0.24 | 1.00/ 1.00 | 0.24 | 1.00/ 1.00 |
| | 100 | 0.00 | 0.00/ 0.30 | 0.00 | 0.00/ 0.30 | 0.00 | 0.00/ 0.30 | 0.73 | 0.30/ 0.30 | 1.46 | 0.70/ 0.70 |
| `dd -ascii` | 10 | 0.00 | 0.00/ 3.00 | 0.00 | 0.00/ 3.00 | 0.00 | 0.00/ 3.00 | 1.43 | 3.00/ 3.00 | 2.50 | 10.00/10.00 |
| | 100 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 0.00 | 0.00/ 0.00 | 0.36 | 0.10/ 0.10 |
| `ls -g` | 10 | 0.00 | 0.00/18.00 | 1.27 | 12.00/25.00 | 1.27 | 12.00/25.00 | 1.42 | 13.00/25.00 | 1.90 | 16.00/28.00 |
| | 100 | 0.00 | 0.00/ 1.00 | 0.31 | 0.30/ 1.30 | 0.31 | 0.30/ 1.30 | 0.63 | 0.40/ 1.30 | 0.78 | 0.40/ 1.30 |
| `ls -l` | 10 | 0.92 | 13.00/28.00 | 2.30 | 24.00/36.00 | 2.30 | 24.00/36.00 | 2.45 | 24.00/36.00 | 3.68 | 26.00/37.00 |
| | 100 | 0.00 | 0.00/ 0.70 | 0.91 | 0.60/ 1.30 | 1.37 | 0.60/ 1.30 | 2.43 | 1.20/ 1.70 | 2.88 | 1.30/ 1.80 |
| `tar` | 10 | 0.00 | 0.00/ 3.00 | 0.09 | 1.00/ 3.00 | 0.09 | 1.00/ 3.00 | 0.17 | 1.00/ 3.00 | 0.61 | 8.00/ 8.00 |
| | 100 | 0.00 | 0.00/ 0.30 | 0.00 | 0.00/ 0.30 | 0.09 | 0.20/ 0.30 | 0.09 | 0.20/ 0.30 | 0.17 | 0.30/ 0.30 |

**Table 4: Percentage of flow origins having history anomalies and traces found as anomaly in test - Using the pruned training tree with threshold 1Sh**

## 6. REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1), 2009.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, pages 263–277, 2008.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 30–40, 2011.

[5] D. Bounov, R. G. Kici, and S. Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, pages 161–176, 2015.

[7] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd Usenix Security Symposium*, pages 385–399, 2014.

[8] S. Designer. “return-to-libc” attack. *Bugtraq, Aug*, 1997.

[9] Ú. Erlingsson. Data-driven software security: Models and methods. In *Proceedings of the 29th Computer Security Foundations Symposium*, pages 9–15, 2016.

[10] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

[11] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd Usenix Security Symposium*, pages 417–432, 2014.

[12] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 11th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013.

[13] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: Size does matter in Turing-complete return-oriented programming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, pages 64–76, 2012.

[14] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, 2014.

[15] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security & Privacy (S&P)*, pages 276–291, 2014.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[17] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 941–951, 2015.

[18] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15st USENIX Security Symposium*, 2006.

[19] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[20] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 577–587, 2014.

[21] B. Niu and G. Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 914–926, 2015.

[22] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, pages 447–462, 2013.

[23] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[24] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2012.

[25] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, pages 745–762, 2015.

[26] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[27] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[28] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, pages 941–955, 2014.

[29] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 927–940, 2015.

[30] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*, 2016.

[31] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[32] J. Walden, J. Stuckman, and R. Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE)*, pages 23–33, 2014.

[33] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, 2012.

[34] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 299–308, 2012.

[35] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu. Shingled graph disassembly: Finding the undecidable path. In *Proceedings of the 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 273–285, 2014.

[36] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*, pages 79–93, 2009.

[37] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. VTrust: Regaining trust on virtual calls. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[38] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Protecting virtual function tables' integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[39] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zo. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, pages 559–573, 2013.

[40] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Conference on Security (USENIX)*, pages 337–352, 2013.

[41] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, pages 421–428, 2010.