Binary Control-Flow Trimming

Masoud Ghaffarinia
The University of Texas at Dallas
ghaffarinia.masoud@utdallas.edu

Kevin W. Hamlen
The University of Texas at Dallas
hamlen@utdallas.edu

ABSTRACT

A new method of automatically reducing the attack surfaces of binary software is introduced, affording code consumers the power to remove features that are unwanted or unused in a particular deployment context. The approach targets stripped binary native code with no source-derived metadata or symbols, can remove semantic features irrespective of whether they were intended and/or known to code developers, and anticipates consumers who can demonstrate desired features (e.g., via unit testing), but who may not know the existence of specific unwanted features, and who lack any formal specifications of the code's semantics.

Through a combination of runtime tracing, machine learning, in-lined reference monitoring, and contextual control-flow integrity enforcement, it is demonstrated that automated code feature removal is nevertheless feasible under these constraints, even for complex programs such as compilers and servers. The approach additionally accommodates consumers whose demonstration of desired features is incomplete; a tunable entropy-based metric detects coverage lapses and conservatively preserves unexercised but probably desired flows. A prototype implementation for Intel x86-64 exhibits low runtime overhead for trimmed binaries (about 1.87%), and case studies show that consumer-side control-flow trimming can successfully eliminate zero-day vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

software debloating, control-flow integrity

ACM Reference Format:
Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *2019 ACM SIGSAC Conference on Computer & Communications Security (CCS'19), November 11–15, 2019, London, United Kingtom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3319535.3345665

1 INTRODUCTION

Security of software is widely believed to be inversely related to its complexity (cf., [83, 93]). With more features, larger implementations, and more behavioral variety come more opportunities for programmer error, malicious code introduction, and unforeseen component interactions.

Unfortunately, economic forces have a history of driving complexity increases in commercial software (sometimes dubbed *Zawinski's law of software envelopment* [65]). Software developers understandably seek to create products that appeal to the widest possible clientele. This "one-size-fits-all" business model has led to commercial software products of increasing complexity, as developers pack more features into each product they release. As a result, software becomes more multi-purpose and more complex, its attack surface broadens, and more potential opportunities for malicious compromise become available to adversaries. For security-sensitive (e.g., critical infrastructure or military) consumers who leave many product features unused but critically rely on others, these security dangers are often unacceptable. Yet because of the market dominance, low cost, and high availability of one-size-fits-all COTS software, bloated software continues to pervade many mission-critical software networks despite the security disadvantages.

As a high-profile example of such feature bloat, in 2014 the bash command interpreter, which is a core component of nearly all Posix-compliant operating systems, was found to contain a series of obscure, undocumented features in its parser [81] that afforded attackers near-arbitrary remote code execution capabilities. Though sometimes referred to as the Shellshock "bug," the vulnerabilities were likely intended as features related to function inheritance when bash was originally written in the 1980s [88]. Their inclusion in a rarely analyzed part of the code caused them to elude detection for a quarter century, exposing millions of security-sensitive systems to potential compromise. This demonstrates that high-complexity software can contain obscure features that may have been *intended by software developers,* but that are unknown to consumers and pose security risks in certain deployment contexts.

Code-reuse attacks [11, 18, 70, 72, 74] are another example of the inherent security risks that code-bloat can introduce. The potential potency of these attacks depends on the variety of code fragments (*gadgets*) in the victim program's executable memory [32, 37], which the attacks abuse to cause damage. Feature-bloated code offers adversaries a larger code-reuse attack surface to exploit. *Control-flow integrity* (CFI) protections [1, 2, 4, 49, 58, 60, 78, 79, 92] defend against such attacks by constraining software to a policy of control-flow graph (CFG) edges that is *defined by the programmer* [2, 79] (e.g., derived from the semantics of the programmer-defined source code, or a recovery of those semantics from the program binary). They therefore do not learn or enforce policies that defend against undocumented feature vulnerabilities like Shellshock, whose control-flows are sanctioned by the source semantics and are therefore admitted by CFI controls.

To demonstrate the openness of this problem, we tested the ability of the 11 source-free CFI solutions listed in Table 1 to automatically mitigate the vulnerabilities listed in Table 2, which each

Table 1: False negative rates of source-free CFI solutions when applied to perform code-debloating

	C-Flat	PathArmor	TypeArmor	Lockdown	BinCC	O-CFI	bin-CFI	CCFIR	MoCFI	NaCl	XFI
false negatives	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Table 2: CVEs of security-evaluated products

Program	CVE numbers
Bash	CVE-2014-6271, -6277, -6278, -7169
ImageMagic	CVE-2016-3714, -3715, -3716, -3717, -3718
Proftpd	CVE-2015-3306
Node.js	CVE-2017-5941
Exim	CVE-2016-1531

constitute an insecure (but possibly developer-intended) functionality, such as Shellshock or ImageTragick (see §5 for details) that was later patched once it became known to consumers. Each algorithm was applied to secure all the binaries against control-flow abuse attacks. All CFI-protected binaries nevertheless remain susceptible to abuse of all the CVEs—a 100% false negative rate. These solutions fail because they were designed to infer and enforce policies that whitelist developer-intended control-flows, not automatically de-bloat hidden features.

To address this unsolved problem, our research introduces *binary control-flow trimming*, a new technology for automatically specializing binary software products to exclude semantic features undesired by consumers, irrespective of whether the features are intended or even known to developers, or whether they are part of a product's source-level design. Control-flow trimming is envisioned as an extra layer of consumer-side defense (i.e., CFG policy tightening) that identifies and excises unwanted software functionalities and gadgets that are beyond the reach of CFI alone.

Learning consumer-unwanted but developer-intended functionalities cannot be achieved with purely non-probabilistic CFG recovery algorithms, such as those central to CFI, because such algorithms approximate a ground truth policy that is is a strict supergraph of the policy needed for feature removal. Tightening that supergraph based on incomplete, consumer-supplied tests requires coupling CFI with trace-based machine-learning. The resulting policy is a more complex, probabilistically constructed, *contextual CFG* (CCFG), which considers fine-grained branch history to distinguish consumer-wanted flows from a sea of developer-intended flows. No prior CFI approach can enforce policies of this complexity because their sensitivity is presently limited to only a few code features (e.g., system API calls [61]), they rely on machine registers or OS modifications unavailable to user code [20, 79], or they require source code access [49] which is often unavailable to consumers. To enforce these policies, we therefore introduce a new contextual CFI enforcement strategy that efficiently encodes contexts as hash codes safely maintainable in user-level machine registers.

In addition, our work assumes that consumers who lack source code probably have no way of formally specifying semantic features or control-flows that they wish to retain, and might not even be aware of all features whose disuse make them candidates for trimming. We therefore assume that consumers merely demonstrate *desired* software features via unit tests (e.g., inputs or user interactions that test behaviors for quality assurance purposes). Such testing is inevitably incomplete and inexhaustive for programs whose input spaces are large (often infinite); so in order to tolerate this incompleteness, we introduce an entropy-based method of detecting points of uncertainty in CCFGs derived from unit tests, and a strategy for relaxing enforcement at such points. Consumers can strengthen the enforcement by supplying additional unit tests that exercise these points more thoroughly.

In summary, we contribute the following:

- 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 binary-only context-sensitive control-flow graph (CCFG) integrity policy formalism derivable from execution traces.
- We propose a machine learning approach to construct CCFGs from runtime trace sets, and demonstrate that it is accurate enough to exhibit a 0% false positive rate for complicated programs such as compilers and web servers.
- We showcase a fully functional prototype that automatically instruments native code with an in-lined reference monitor (IRM) [68] that enforces the CCFG policy.
- Experiments show that control-flow trimming can eliminate zero-day vulnerabilities associated with removed functionalities, and that the approach exhibits low runtime overheads of about 1.87%.

Section 2 first gives a high level overview of our system. Sections 3 and 4 next detail our technical approaches to feature identification and policy enforcement for Intel x86-64 native codes, respectively. Section 5 evaluates the approach in terms of accuracy and performance, followed by a discussion of limitations and future work in Section 6. Finally, Section 7 compares our work with related research, and Section 8 concludes.

2 APPROACH OVERVIEW

2.1 Contextual Control-flow Graph Policies

Our approach assumes that feature-trimming specifications are informal, taking the form of unit tests that exercise only the consumer-desired features of the software. Such testing is commonly practiced by security-sensitive consumers. One obvious approach to trimming unwanted features entails simply erasing all code bytes that remain unexecuted by the tests. However, our early experimentation taught us that this blunt approach fails for at least two reasons: (1) It requires an unrealistically comprehensive unit test set, lest some code bytes associated with wanted features go unexercised and get improperly erased. Such comprehensive testing is very difficult to achieve without source code. (2) It often retains unwanted features due to the modular design of complex software, which reuses each individual code block to implement multiple semantic features—some wanted and some unwanted. When all code blocks for an unwanted feature are each needed by some wanted feature, the unwanted feature cannot be trimmed via code byte erasure without corrupting the wanted features.

These experiences led us to adopt the more general approach of control-flow trimming. Control-flow trimming removes semantic features by making the control-flow paths that implement the feature unreachable—e.g., by instrumenting all computed jump instructions in the program with logic that prohibits that flow. This generalizes the code byte erasure approach because, in the special case that the trimmed CFG contains no edges at all to a particular code block, that block can be erased entirely.

We also discovered that control-flow policies that successfully distinguish consumer-undesired (yet developer-intended) code features from consumer-desired features tend to be significantly more complex and powerful than any prior CFI solution can efficiently enforce. In particular, policy decisions must be highly context-sensitive, considering a detailed *history* of prior CFG edges traversed by the program in addition to the next branch target when deciding whether to permit an impending control transfer. Since trace histories of real-world programs are large (e.g., unbounded), these decisions must be implemented in a highly efficient manner to avoid unreasonable performance penalties for the defense.

To illustrate, assume critical functionality F_1 executes code blocks $c_1; c_2; c_3; c_4$ in order, whereas undesired functionality F_2 executes $c_1; c_3; c_3; c_4$. A strict code byte erasure approach cannot safely remove any blocks in this case, since all are needed by F_1. However, control-flow trimming can potentially delete CFG edges (c_1, c_3) and (c_3, c_3) to make functionality F_2 unrealizable without affecting F_1.

Extending the example to include context-sensitivity, consider an additional critical functionality F_3 implemented by sequence $c_2; c_3; c_3; c_1; c_3; c_4$. This prevents removal of edges (c_1, c_3) and (c_3, c_3) from the CFG, since doing so would break F_3. But an enforcement that permits edge (c_3, c_3) conditional on it being immediately preceded by edge (c_2, c_3) successfully removes F_2 without harming F_3. In general, extending the context to consider the last n edges traversed lends the technique greater precision as n increases, though typically at the cost of higher space and time overheads. A balance between precision and performance is therefore needed for best results, which we explore in §5.

2.2 Automated, In-lined CCFG Enforcement

Figure 1 depicts our control-flow trimming architecture. The input to our system consists of stripped x86-64 binaries along with sample execution traces that exercise functionalities wanted by the consumer. The rewriter automatically disassembles, analyzes, and transforms them into a new binary whose control-flows are constrained to those exhibited by the traces, possibly along with some additional flows that could not be safely trimmed due to uncertainty in the trace set or due to performance limitations. We assume that no source code, debug symbols, or other source-derived metadata are provided. Prior work on *reassemblable disassembly* [86] has established the feasibility of recovering (raw, unannotated) assembly

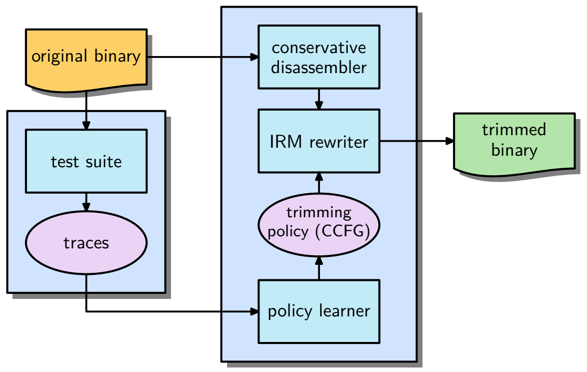

Figure 1: Binary control-flow trimming system architecture

files from binaries for easier code transformation, allowing us to use assembly files as input to our prototype during evaluations (§5).

Discerning a consumer-desired CCFG policy based on traces without access to sources is challenging. Our approach applies machine learning to traces generated from the test suite to learn a subgraph of the developer-intended flows. The output of this step is a decision tree forest, with one tree for each control-flow transfer point in the disassembled program. Each decision tree consults the history of immediately previous branch destinations, along with the impending branch target, to decide whether to permit the impending branch. The forest therefore defines a CCFG policy.

Since decision trees tend to overfit the training, it is important to detect overfitting and relax the policy to permit traces that were not exhibited during training, but whose removal might break consumer-desired functionalities. We therefore assign an entropy-based confidence score to each node of the decision forest. Nodes with unacceptable confidence receive relaxed enforcement by pruning their children from the tree. In the extreme case, pruning all trees to a height of 1 results in a non-contextual CFG that matches the policy enforced by most non-contextual (backward- and forward-edge) CFI. Trimming therefore always enforces a policy that is at least as strict as non-contextual CFI, and usually stricter.

After deriving a suitable CCFG, the policy is enforced via in-lined reference monitoring. Specifically, we surround each control-flow transfer instruction in the program with guard code that maintains and updates a truncated history of branch targets expressed as a hash code. A read-only hash table determines whether the impending branch is permitted. Policy-violating branches yield a security violation warning and premature termination of the program.

2.3 Threat Model

Like prior research on CFI and artificial diversity, success of our approach can be measured in terms of two independent criteria: (1) inference of an accurate policy to enforce, and (2) enforcement of the inferred policy. For example, COOP attacks [69] exploit lapses in the first criterion; they hijack software by traversing only edges permitted by the policy, which is insufficiently precise. In contrast, coarse-grained CFI approaches are susceptible to lapses in the second criterion; to achieve high performance, they enforce a policy approximation, which sometimes allows attackers to exploit approximation errors to hijack the code (e.g., [17]). Artificial diversity defenses can experience similar failures, as in the case of implementation disclosure attacks [10, 24, 31, 71, 73].

With regard to the first criterion, our approach is probabilistic, so success is evaluated empirically in §5 in terms of false negatives and false positives. (The false classification rates measure accuracy against a policy that differs from CFI policies, however, since control-flow trimming has a stricter model of ground truth than CFI, as described in §1.) With regard to the second criterion, we assume a relatively strong threat model in which attackers have complete read-access to the program image as it executes, and even have write-access to all writable data pages, but lack the power to directly change page access permissions. Thus, attackers know the policy being enforced but lack the ability to change it since its runtime encoding resides in read-only memory. (We assume that DEP or W$\bigoplus$X protections prevent writes to code and static data sections.) Attackers also cannot directly corrupt CPU machine registers, affording our defense a safe place to store security state.

Since our defense enforces a control-flow policy, non-control data attacks are out of scope for this work. We defer mitigations of such attacks to other defense layers.

3 DETAILED DESIGN

3.1 Learning CCFG Policies

Since it is usually easier for code-consumers to exhibit all features they wish to preserve (e.g., through software quality testing), rather than discovering those they wish to remove, we adopt a whitelisting approach when learning consumer control-flow policies:

A *trace* $e_1, e_2, e_3, \ldots$ is defined as the sequence of control-flow edge traversals during one run of the program, where e_i is the ith edge taken. We include in the edge set all binary control-flow transfers except for unconditional branches and fall-throughs of non-branching instructions (whose destinations are fixed and therefore not useful to monitor). Thus, the edge set includes targets of conditional branches, indirect (computed) branches, and returns.

Let T_1 be a set of program execution traces that exhibit only software features that must be preserved, and let T_2 be a set that includes traces for both wanted and unwanted features. T_1 is provided by the user, and is assumed to be noise-free; every trace exhibited during training is a critical one that must be preserved after control-flow trimming. However, we assume there may be additional critical traces requiring preservation that do not appear in T_1. The learning algorithm must therefore conservatively generalize T_1 in an effort to retain desired functionalities. T_2 is assumed to be unavailable during training, and is used only for evaluation purposes to assess whether our training methodology learns accurate policies.

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

Context lengths are not fixed in our model. While an upper bound on context lengths is typically established for practical reasons, our approach considers different context lengths at different

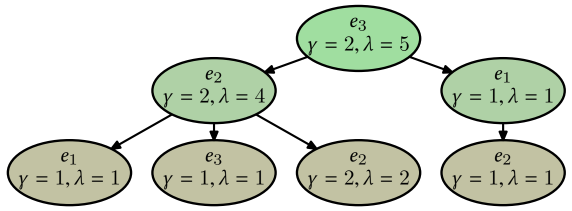
e_3
$\gamma = 2, \lambda = 5$
e_2
$\gamma = 2, \lambda = 4$
e_1
$\gamma = 1, \lambda = 1$
e_1
$\gamma = 1, \lambda = 1$
e_3
$\gamma = 1, \lambda = 1$
e_2
$\gamma = 2, \lambda = 2$
e_2
$\gamma = 1, \lambda = 1$

Figure 2: Sample decision tree for edge e_3

branch sites based on an estimate of the benefits, as measured by information gain. In our design, we first suppose there is a fixed size (possibly large) for the contexts, and then proceeded to accommodate variable-sized contexts.

To maximize effectiveness, contexts must include as much policy-relevant control-flow information as possible without being polluted with uninformative edges. Indirect branches and returns are the primary sources of control-flow hijacks, so are included. Direct calls and jumps are also included even though they have fixed destinations, because we found that doing so allows the training to learn a form of call-return matching that improves accuracy. We also include conditional branch destinations in the contexts, since they often implement series of tests that conditionally activate software features that may be targets of trimming.

The learning algorithm is a binary classification that decides for each control-flow edge whether it is permissible, based on the last k edges currently in the context. We chose decision trees as our learning model, since they are relatively simple and efficient to implement at the binary level. While decision trees can suffer from overfitting, such overfitting is potentially advantageous for our problem because every trace in T_1 must be preserved. Higher security therefore results from a conservatively tight model that can be conditionally relaxed at points of uncertainty.

For a given edge e, the learning algorithm creates a decision tree as follows: The root is labeled with e and the depth of the tree is k, where k is the maximum size of the context. Each node at level $i \geq 1$ of the tree is labeled with the edge e' appearing immediately before the context defined by the path from the node's parent at level i up to the root. It is additionally annotated with the number of traces γ and number of contexts λ in which that particular edge-label occurs at that context position. These numbers are used during uncertainty detection and policy relaxation (§3.2).

Every leaf of this tree represents a permissible control-flow history encoded by the path from it to the root. The feature encoded by a node at level $i + 1$ is the i-to-last edge in the context when the edge labeled at the root is reached. So, given a context χ we can check whether it is permissible as follows: The last edge in χ must be a root of some tree in our learned decision tree forest; otherwise the impending branch is rejected. The penultimate edge in χ should be one of that root's children; otherwise the impending branch is rejected. We continue to check the context edges in χ in reverse order until we reach a decision tree leaf. Reaching a leaf implies policy-compliance, and the impending branch is permitted.

To illustrate, consider a hypothetical program with two sample traces: one containing sub-sequences $[e_1, e_2, e_3]$, $[e_2, e_2, e_3]$ and $[e_3, e_2, e_3]$; and the other containing sub-sequences $[e_2, e_1, e_3]$ and $[e_2, e_2, e_3]$. Figure 2 shows the decision tree made for edge e_3 out of these sub-traces. The root is labeled with $(e_3, \gamma = 2, \lambda = 5)$, since there are 2 traces and 5 histories having edge e_3. Edge e_2 is the penultimate edge in 4 of those cases, and e_1 is the penultimate edge in 1 case, causing nodes $(e_2, \gamma = 2, \lambda = 4)$, and $(e_1, \gamma = 1, \lambda = 1)$ to comprise the next level of the tree. In the same way, the nodes at the bottom level correspond to the antepenultimate edges appearing in each context. Edges e_1, e_3, and e_2 are antepenultimate when e_2 is penultimate, and e_2 is antepenultimate when e_1 is penultimate. Observe that the labels are not unique; the same label or edge can be assigned to some other node of the same tree. In addition, for any node, λ is the sum of its child λ's, while γ is not.

3.2 CCFG Policy Relaxation

To cope with the inevitable incompleteness of training data that is assumed to be amassed without guidance from source code, we next consider the problem of generalizing the decision tree forest to include more child nodes than were explicitly observed during training. In general, if training observes many diverse jump destinations for a specific subtree, that subtree may have a complex behavior that was not exhaustively covered by training. There is therefore a high chance that additional consumer-desired destinations for that branch site exist that were not explicitly observed.

The same is true for diverse collections of contexts. If the contextual information at a given tree node is highly diverse and offers little information gain, this indicates that the context at that position is not a useful predictor of whether the impending branch is permissible. For example, the branch may be the start of what the user considers an independent semantic feature of the software, in which case the context is reflecting a previous semantic feature that has little relevance to the permissibility of this branch point. Thus, nodes with numerous low-frequency child nodes should be considered with low confidence.

To estimate this confidence level, we use entropy to calculate an uncertainty metric using the number of times different child nodes of a node appear in the training. Nodes with diverse children have higher entropy. The confidence score of a node n is computed as

$$confidence(n) = \frac{\gamma}{N} \times -\frac{1}{M^2} \sum_{m=1}^{M} \frac{\lambda_m}{\lambda} \log_M\left(\frac{\lambda_m}{\lambda}\right) \tag{1}$$

where (e, γ, λ) is node n's label, M is the number of node n's children, $(e_m, \gamma_m, \lambda_m)$ is child m's label, and N is the total number of traces.

This formula combines the probability of a node being in a trace, the entropy of its children λ, and the number of its children. It is inversely related to entropy because, for any given number of children M, we have higher confidence if the distribution of child frequencies is relatively flat. For example, if we observe two children with λ's 5 and 5, we have higher confidence than if we observe two children with λ's 1 and 9. The former indicates a well-covered, predictable behavior, whereas the latter is indicative of a behavior with rare outliers that were not covered well during training. Fewer children likewise engender higher confidence in the node.

An ideal confidence threshold t^* that maximizes accuracy on the training set is computed using crossfold validation (see §5), and all children with confidence below t^* are pruned from the forest. In the worst case, pruning all the trees to a height of 1 yields a non-contextual CFG that is the policy that would be enforced by typical non-contextual CFI (i.e., no debloating). Pruning therefore finds a middle ground between trimming only the developer-unintended features and over-trimming the consumer-wanted features.

For example, in Figure 2 the confidence score of the root and the node labeled $(e_2, \gamma = 2, \lambda = 4)$ are 0.36 and 0.31, respectively. If our confidence threshold exceeds a node's confidence score, then context is disregarded when making policy decisions at that origin. So in our example, a confidence threshold of 0.35 prunes the tree after node $(e_2, \gamma = 2, \lambda = 4)$, making that node a leaf. This refines the policy by disregarding policy-irrelevant context information.

3.3 Enforcing CCFG Policies

In-lining guard code that enforces a highly context-sensitive policy at every computed branch without incurring prohibitive overheads raises some difficult implementation challenges. To track and maintain contexts, our enforcement must additionally instrument all direct calls, conditional branches, and interrupt handlers with context-update logic. Space-efficiency is a challenge because CCFG policies are potentially large—code with b branch sites and context-length bound k can have CCFG policies of size $O(b^k)$ in the worst case. Time-efficiency is a challenge because policy decisions for CCFGs potentially require $O(k)$ operations, in contrast to non-contextual CFG policies, which engender constant-time decisions.

To obtain acceptable overheads in the face of these challenges, our implementation compactly represents contexts as hash codes, and represents CCFG policies as sparse hash tables of bits, where an entry of 1 indicates a permitted context. The hash function need not be secure since our enforcement protects hash values via access controls (see §4), but it must be efficiently computable and uniform. We therefore use the relatively simple hash function given by

$$hash(\chi) = \bigoplus_{i=1}^{|\chi|} ((\pi_2 \chi_i) \ll (|\chi| - i)s) \tag{2}$$

where $\bigoplus$ is xor, $|\chi|$ is the length of context χ, $\pi_2 \chi_i$ is the destination (second projection) of the ith edge in χ, $\ll$ is bit-shift-left, and $s \geq 0$ is a shift constant. This has the advantage of being computable in an amortized fashion based on the following recursion:

$$hash(\chi e) = (hash(\chi) \ll s) \oplus (\pi_2 e) \tag{3}$$

The CCFG hash table is constructed by storing a 1 at the hash of every policy-permitted context. This can introduce some imprecision in the form of hash collisions, since a policy-violating context can have the same hash code as a policy-permitted context, causing both to be accepted. However, this collision rate can be arbitrarily reduced by increasing shift-constant s and the bit-width w of shift operation $\ll$. For example, setting s to the address-width a and using $w = ka$ guarantees no collisions, at the expense of creating a large table of size 2^{ka-3} bytes. On 64-bit architectures, we found that using $s = 1$ and $w \approx \log_2 c$ where c is the code segment size works well, since all branch destination *offsets* (into their respective code segments) are less than c, and the offset portion of the address is where the most policy-relevant bits reside. This yields a hash table of size $O(c)$, which scales linearly with program size.

Table 3: Guard checks for each kind of branch type

Description	Original code	Rewritten Code

Conditional Jumps	`jcc l`	`call jcc_fall`
		`.quad l`

Indirect calls	`call r/[m]`	`mov r/[m], %rax`
		`call indirect_call`

Indirect Jumps	`jmp r/[m]`	`mov %rax, -16(%rsp)`
		`mov r/[m], %rax`
		`call indirect_jump`

Variable Returns	`ret n`	`pop %rdx`
		`lea n(%rsp), %rsp`
		`push %rdx`
		`jmp return`

Returns	`ret`	`mov (%rsp), %rdx`
		`jmp return`

4 IMPLEMENTATION

To generate sample traces, we use Pin [46] and DynamoRIO [14] to track all branches during each run of each test program. We then apply our machine learning algorithm with the hash function defined in §3.3 to generate the CCFG hash table. The hash table is added in a relocatable, read-only data section accessible from shared libraries while protecting it from malicious corruption.

Table 3 transforms each type of branch instruction (column 2) to guard code (column 3). To reduce code size overhead, the guard code is modularized into trampolines that jump to a policy-check before jumping to each target. This trades smaller code size for slightly higher runtime overhead. Table 4 shows the details of the trampoline code called by branch guards (Table 3), which invoke policy checks and state updates (Table 5).

Guard code for conditional jumps must carefully preserve all CPU status flags until the branch decision is made. Since sequences of n consecutive conditional jumps can implement an n-way branch, we avoid corrupting status flags by updating the context before the sequence is complete, in-lining only one fall-through trampoline for the sequence. This is achieved by using another trampoline `jcc_back` for the first $n-1$ instructions, which fall-through without checking the destination because the guards in Table 5 are the only parts that affect flags. A similar strategy applies to conditional branches followed by Intel conditional-moves (`setcc` and `cmovcc`). This results in a maximum of 67 trampolines for all possible conditional jumps (2×32 for the two directions of each of the 32 possible conditional jump instructions on x86-64, plus 3 other trampolines `fall_l`, `back_l`, and `jump_l`).

Table 5 shows the common guard invoked by the trampolines, which updates the context and consults the hash table to enforce the policy. Two implementations are provided: the center column uses SSE instructions, which are widely available on Intel-based processors; while the rightmost column provides a more efficient implementation that leverages SHA-extensions (`sha1msg1` and `sha1msg2`) that are presently only available on a few processor lines [5]. Our experiments and the descriptions that follow use the legacy-mode implementation, but we expect improved performance of our algorithm as SHA extensions become more available.

Table 4: Trampolines used in guards referred in Table 3

Label	Assembly Code

`indirect_jump:`	`push %rax`
	`common-guard`
	`mov -8(%rsp), %rax`
	`ret`

`indirect_call:`	`push %rax`
	`common-guard`
	`ret`

`return:`	`common-guard` `ret`

`jcc_fall:`	`jcc jump_l`
	`jmp fall_l`

`jcc_back:`	`jcc jump_l`
	`jmp back_l`

`jump_l:`	`xchg (%rsp), %rax`
	`mov (%rax), %rax`
	`jmp condition_jump`

`fall_l:`	`xchg (%rsp), %rax`
	`lea 8(%rax), %rax`
	`jmp condition_jump`

`back_l:`	`xchg (%rsp), %rax`
	`lea 8(%rax), %rax`
	`xchg (%rsp), %rax`
	`ret`

`condition_jump:`	`push %rax`
	`common-guard`
	`pop %rax`
	`xchg (%rsp), %rax`
	`ret`

For efficiency and safety, we store contexts in 128-bit `xmm` registers rather than memory. Register `%xmm14` maintains a length-4 context as four packed 32-bit unsigned integers, and `%xmm15` maintains the context hash. On entry to the `before-check` code, `%xmm13` contains the section base address and general (64-bit) register r holds the impending branch target to check. Register r varies depending on the branch type (`%rdx` for returns and `%rax` for others).

This implementation strategy requires the target program to have at most 12 live `xmm` registers (out of 16 total) at each program point, leaving at least 2 to globally maintain context and context-hash, plus 2 more for scratch use at each guard site. More constrained `xmm` register usage is rare, but can be supported by spilling `xmm` registers to general-purpose registers or to memory. Two of the evaluated programs in §5 require this special treatment (postgres and postmaster), and exhibited slightly higher than average overheads of 3% as a result.

Lines 1–2 of `before-check` calculate the target offset. Line 3 then updates the hash code using Equation 3. After this, `%xmm12` and `%xmm15` have the target offset and the new hash, respectively.

The `check` operation implements the policy check. Line 5 truncates the hash value to the size of the hash table. Finally, line 6 finds the bit corresponding to the hash value in the table, and line 7 jumps to the trap in case it is unset, indicating a policy rejection.

The `after-check` code updates the history in `%xmm14` and the hash code in `%xmm15`. It does so by extracting the oldest context entry about to be evicted (line 8), shifting the context left to evict the oldest entry and make space for a new one (line 9), adding the new entry (line 10), and leveraging involutivity of xor to remove the evicted entry from the hash code (lines 11–12). Finally, lines 13–14 left-shift the context and hash code by one bit in preparation for the next context and hash update.

Table 5: Guard checks implementation for trampolines referred as `common-guard` in Table 4

Guard Name	Legacy-mode		SHA-extension	
	Guard Code			
before-check	1:movd	r, %xmm12	1:movd	r, %xmm12
	2:psubd	%xmm13, %xmm12	2:psubd	%xmm13, %xmm12
			3:sha1msg1	%xmm14, %xmm15
			4:sha1msg2	%xmm15, %xmm15
			5:pslrdq	$4, %xmm15
	3:pxor	%xmm12, %xmm15	6:pxor	%xmm12, %xmm15
check	4:movd	%xmm15, r	7:movd	%xmm15, r
	5:and	$max\_hash - 1$, r	8:and	$max\_hash - 1$, r
	6:bt	r, (HASH_TABLE)	9:bt	r, (HASH_TABLE)
	7:jnb	TRAP	10:jnb	TRAP
after-check	8:pextrd	$3, %xmm14, r	11:pslldq	$4, %xmm14
	9:pslldq	$4, %xmm14	12:psllw	$1, %xmm14
	10:pxor	%xmm12, %xmm14	13:pxor	%xmm12, %xmm14
	11:movd	r, %xmm12		
	12:pxor	%xmm12, %xmm15		
	13:pslld	$1, %xmm15		
	14:pslld	$1, %xmm14		

One important deployment consideration is whether to exclude library control-flows from the program flow, since they are shared, and it may therefore be infeasible to learn appropriate policies for them based on profiling only some applications that load 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 enforce the policy on all branches from any portion of the program code section and all the shared libraries shipped with it, but we leave system shared libraries unaltered. The latter can optionally be trimmed by making a local copy to which the policy is applied, though the result is obviously no longer a library that can be shared across multiple applications.

When rewriting app-included shared libraries, we add trampolines to each image, and declare them with `.hidden` visibility to avoid symbol name-clashes between the images. The hash table can be specialized to each image or centralized for all. For this work we use one centralized table for all interoperating images, accessed via the `.got` table for concurrent, shared access between modules.

5 EVALUATION

We experimentally evaluated our control-flow trimming system in terms of performance, security, and accuracy. Performance evaluation measures the overhead that our system imposes in terms of space and runtime. Our security analysis examines the system's ability to withstand the threats modeled in §2.3. Security failures therefore correspond to false negatives in the classification. Finally, accuracy is measured in terms of false positives—premature aborts of the trimmed program when no policy violation occurred.

Test programs consist of the real-world software products in Table 6, plus bash, gcc, ImageMagic, the epiphany and uzbl browsers, and the SPEC2017 benchmarks. We also successfully applied our prototype to rewrite the full GNU Coreutils 8.30 collection. The browsers were chosen primarily for their compatibility with Pin and DynamoRIO, which we use for trace collection and replay.

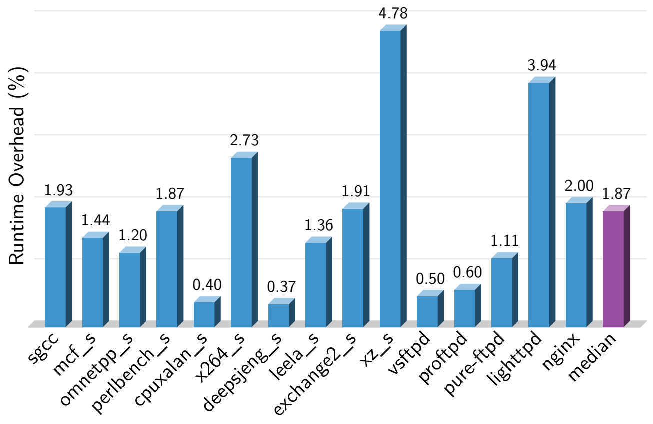

Figure 3: Runtime overhead for SPEC2017 intspeed suite and some ftp- and web-servers

To evaluate accuracy, we created or obtained test suites for each program. For example, in the gcc evaluations, we used the gcc source code as its own input for unit testing. That test suite therefore consists of all C source files needed to compile gcc on the the experiment machine. For ImageMagic, we randomly gathered hundreds of JPEG and PNG images. We unit-tested ftp servers by downloading and uploading randomly selected files interspersed with random ftp commands (e.g., cd, mkdir, ls, append, and rename). For exim we used a script to launch sendmail and randomly send an email to a specific address. Browser experiments entail loading pages randomly drawn from the Quantcast top 475K urls, and uzbl experiments additionally include random user interactions (e.g., back/forward navigation, scrolling in all directions, zoom in/out, search, etc.). All results were obtained using a DELL T7500 machine with 24G of RAM and Intel Xeon E5645 processor.

5.1 Performance Overhead

Figure 3 graphs the runtime overhead for SPEC2017 benchmarks and several ftp- and web-servers. We used Apache benchmark [8] to

Table 6: Space overhead for SPEC2017 intspeed suite benchmarks and some real-world applications

	Original Size (KB)		**Size Increase** (%)	
Binary	**File**	**Code**	**File**	**Code**
perlbench_s	10686	1992	10.17	35.14
sgcc	63243	8499	12.76	59.15
mcf_s	131	19	8.80	35.20
omnetpp_s	28159	1567	5.15	55.37
cpuxalan_s	80762	4701	4.19	48.25
x264_s	3320	567	6.41	23.40
deepsjeng_s	508	85	10.23	42.17
leela_s	3819	191	2.15	45.14
exchange2_s	182	111	16.01	18.61
xz_s	1082	146	0.69	2.12
exim	1407	1187	32.14	14.70
lighttpd	1304	294	13.12	27.12
memcached	746	156	13.50	23.89
nginx	1674	1444	29.76	19.07
openssh	2467	638	15.12	21.40
proftpd	3310	803	16.34	29.12
pureftpd	470	118	17.12	27.04
vsftpd	143	133	25.78	28.99
postgresrl	757	544	41.35	33.53
node.js	36758	30059	28.63	17.84
median	1541	556	16.42	28.06

issue 25,000 requests with concurrency level of 10 for benchmarking lighttpd and nginx. To benchmark the FTP servers, we wrote a Python script based on the pyftpdlib benchmark [66] to make 100 concurrent clients, each of which request 100 1KB-sized files.

The median runtime overhead is 1.87%, and all benchmarks exhibit an overhead of 0.37–4.78%. The good performance is partially attributable to Table 5's reliance on SIMD instructions, which tend to exercise CPU execution units independent of those constrained by the mostly general-purpose instructions in the surrounding code. This allows out-of-order execution (OoOE) hardware optimizations in modern processors [39] to parallelize many guard code μops with those of prior and subsequent instructions in the stream.

Table 6 shows the space overhead for the SPEC2017 benchmarks and a sampling of the other tested binaries. On average, the test binaries increase in size by 16.42% and their code sizes increase by 28.06%. The main size contributions are the extra control-flow security guard code in-lined into code sections, and the addition of the hash table that encodes the CCFG policy.

Although these size increases are an important consideration for memory and disk resources needed to support our approach, we emphasize that they are not an accurate measure of the resulting software attack surface, since many of the added bytes are non-executable or erased (exception-throwing) opcodes (e.g., `int3`). Attack surface must therefore be measured in terms of reachable code bytes, not raw file or code section size.

To evaluate this, Table 7 measures the reachable, executable code from the decision trees for binaries with a test suite. Despite the increase in total file and code sizes, the amount of reachable code is reduced by an average of 36%. For example, the attack surface of ImageMagic convert is reduced by 94.5%. (The method of computing Table 7 is detailed in §5.3.)

5.2 Security

5.2.1 Vulnerability Removal. A primary motivation for control-flow trimming is the possible removal of defender-unknown vulnerabilities within code features of no interest to code consumers. To test the efficacy of our approach for removing such zero-days, we tested the effects of control-flow trimming on unpatched versions of Bash 4.2, ImageMagic 6.8.6–10, Proftpd 1.3.5, Node.js 8.12, and Exim 4.86 that are vulnerable to the CVEs shown in Table 2, including Shellshock and ImageTragick.

Shellshock attacks exploit a bug in the bash command-line parser to execute arbitrary shellcode. The bug erroneously executes text following function definitions in environment variables as code. This affords adversaries who control inputs to environment variables remote code execution capabilities. Because of its severity, prevalence, and the fact that it remained exploitable for over 20 years before it was discovered, Shellshock has been identified as one of the highest impact vulnerabilities in history [25].

ImageMagick is used by web services to process images and is also pre-installed in many commonly used Linux distributions such as Ubuntu 18.04. ImageTragick vulnerabilities afford attackers remote code execution; delete, move, and read access to arbitrary files; and server-side request forgery (SSRF) attack capabilities in ImageMagic versions before 6.9.3–10, and in 7.x before 7.0.1-1.

ProFTPD 1.3.5 allows remote attackers to read and write from/to arbitrary files via `SITE CPFR` and `SITE CPTO` commands. In node serialize package 0.0.4, the `unserialize` function can be exploited by being passed a maliciously crafted JS object to achieve arbitrary code execution. Exim before 4.86.2 allows a local attacker to gain root privilege when Exim is compiled with Perl support and contains a `perl_startup` configuration variable.

Unit tests for the bash experiment consist of the test scripts in the bash source package, which were created and distributed with bash before Shellshock became known. The tests therefore reflect the quality assurance process of users for whom Shellshock is a zero-day. For the remaining programs, we manually exposed each to a variety of inputs representative of common usages. For ImageMagic, our unit tests execute the application's convert utility to convert images to other formats. We unit-tested ProFTPD by exposing it to a variety of commands (e.g. `FEAT`, `HASH`), excluding the `SITE` command. For Node.js we wrote some JS code that does not leverage node-serialize package. We ran Exim without a `perl_startup` configuration variable.

Using these test suites, we applied the procedure described in §3 to learn a CCFG policy for these five vulnerable programs, and automatically in-lined an enforcement of that policy approximated as a bit hash table. No source code was used in any of the experiments.

Control-flow trimming these programs with these test suites has the effect of removing all the listed vulnerabilities. For example, Shellshock-exploiting environment variable definitions push bash's control-flow to an obscure portion of the parser logic that is trimmed by the learned CCFG policy, and that the in-lined guard code therefore rejects. Similar policy rejections occur when attempting to trigger the vulnerabilities in the other binaries. This demonstrates that control-flow trimming can effectively remove zero-days if the vulnerability is unique to a semantic feature that remains unexercised by unit testing.

Table 7: False positive ratios (%). Zero threshold means no pruning (most conservative) (§3.2).

			Context Anomalies			Origin Anomalies			Trace Anomalies			Reachable
Program	**Samples**	**t***	**t=0.00**	**t=0.25**	**t=t***	**t=0.00**	**t=0.25**	**t=t***	**t=0.00**	**t=0.25**	**t=t***	**Code (%)**
proftpd	10	0.48	3.04	2.37	1.75	4.51	3.95	2.81	45.00	30.00	25.00	47.31
	100	0.37	0.43	0.17	0.05	1.68	1.02	0.37	3.00	1.50	1.00	47.81
	500	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	47.85
vsftpd	10	0.38	2.45	2.16	1.60	3.74	3.23	1.80	35.00	25.00	25.00	51.11
	100	0.23	0.33	0.07	0.14	0.91	0.17	0.22	2.00	1.50	1.50	51.47
	500	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	51.47
pure-ftpd	10	0.41	2.23	1.96	1.43	3.61	3.14	2.83	25.00	25.00	10.00	49.89
	100	0.28	0.04	0.00	0.00	0.15	0.00	0.00	2.50	1.50	1.50	50.03
	500	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	50.05
exim	10	0.25	2.72	1.12	1.88	5.12	4.06	4.81	35.00	15.00	20.00	10.31
	100	0.53	0.58	0.01	0.00	1.36	0.01	0.00	7.50	1.00	0.00	10.63
	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	10.65
ImageMagic convert	10	0.64	0.21	0.10	0.04	1.51	1.23	0.91	20.00	15.00	10.00	5.27
	100	0.54	0.09	0.07	0.00	0.17	0.10	0.00	2.50	1.00	0.00	5.53
	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	5.55
gcc	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	7.66
	100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	7.66
	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	7.66
epiphany	10	0.93	10.91	0.22	0.00	19.60	1.29	0.00	85.00	40.00	0.00	23.41
	100	0.81	10.76	0.20	0.08	15.50	1.14	0.57	40.00	10.00	6.50	23.73
	500	0.33	2.94	0.01	0.01	12.14	0.09	0.08	8.70	0.40	0.30	24.01
	1000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	24.01
uzbl	10	0.92	2.16	0.25	0.12	18.90	1.30	0.81	90.00	40.00	30.00	30.81
	100	0.83	2.09	0.04	0.03	17.36	0.96	0.75	50.50	3.50	2.50	30.83
	500	0.65	0.57	0.01	0.01	9.08	0.34	0.17	10.70	0.90	0.60	30.91
	1000	0.45	0.46	0.03	0.02	7.94	0.52	0.33	4.30	0.85	0.35	30.91

Unit-tested features of the test programs all remain functional after CCFG learning and enforcement. Section 5.3 evaluates the accuracy more precisely by measuring false positive rates under a variety of conditions.

5.2.2 Gadget Analysis. Although control-flow trimming is primarily envisioned as a semantic feature removal method and not a gadget removal method, gadget chains are nevertheless one example of a class of unwanted semantic features that control-flow trimming might remove. To study the effect of control-flow trimming on gadget reachability, we used ROPgadget [67] to find all gadgets in the rewritten test binaries. Since our threat model pessimistically assumes attackers have unrestricted write-access to the stack and heap, our gadget reachability analysis assumes that the attacker can cause any of these gadget addresses to flow as input to any indirect branch or return instruction in the original program. Our defense substitutes all such instructions with guarded-branches and replaces unreachable instructions with `int3`; thus, in order to circumvent (i.e., jump over) the guards to reach a hijackable instruction, the attacker must first supply at least one malicious gadget address that the guards accept, in order to initiate the chain.

To evaluate whether this is possible, for each program we collected all contexts of length $k - 1$ observed during training and testing, appended each discovered gadget address to each, and computed the hashes of the resulting length-k contexts. We then examined whether the hash table that approximates the CCFG policy contains a 0 or 1 for each hash value. In all cases, the hash table entry was 0, prompting a policy rejection of the hijacking attempt. We also simulated the attacks discovered by ROPgadget using Pin and verified that the guards indeed block the attacks in practice.

We can also study the theoretical probability of realizing a gadget chain. The probability of finding a gadget address that can pass the guard code to initiate a gadget chain is approximately equal to the ratio p of 1's to 0's in the hash table that encodes the CCFG policy. This can be reduced almost arbitrarily small by increasing the hash table size relative to the code size (see §3). For example, in gcc this ratio is as small as 0.004. Only 650KB of the original 8499KB code section is visited by the unit tests and remains reachable after control-flow trimming—an attack surface reduction of 92%.

Moreover, if control-flow trimming is coupled with software fault isolation (SFI) [50, 82, 89] to enforce indivisible basic blocks for the guarded-jump trampolines in Table 4, then the probability of realizing a length-n gadget chain reduces to p^n. Since SFI is much easier to realize than CFI for source-free binaries (because it enforces a very simple CFG recoverable by binary disassembly), and tends to impose very low runtime overhead, we consider such a pairing to be a promising direction of future work.

5.3 Accuracy

5.3.1 Specificity. To measure our approach's accuracy in retaining consumer-desired features while excluding undesired ones, we used the programs in Table 7, including several real-world ftp servers, exim, ImageMagic convert, gcc, and two web browsers, since they constitute large, complex pieces of software.

To test gcc, we trained by compiling its own source code to a 64-bit binary, and tested by attempting to compile many C programs to various architectures (32-bit and 64-bit) using the trimmed binary.

For other programs we used the test suites described earlier. In the ImageMagic experiments, the desired functionality is converting a JPG picture to PNG format, and the undesired functionality is

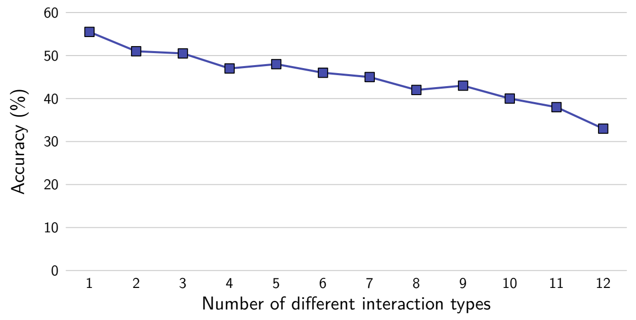

Figure 4: Accuracy vs. interaction diversity with uzbl, using a fixed training set size of 100 and $t = 0.0$

resizing a picture. For ftp servers, the undesired functionalities are the SITE and DELETE commands, and the remaining commands are desired. Ftp file content and command order were randomized during training and evaluation. For exim, the undesired functionality is -oMs (which sets the sender host name instead of looking it up). The undesired functionalities for epiphany and uzbl-browser are incognito mode and cookie add/delete, respectively.

Positives in the classification are execution failures during testing, as signaled by premature abort with a security violation warning. *False negatives* are runs that exercise a consumer-undesired semantic feature even after trimming. In contrast, a *false positive* occurs when the defense aborts a consumer-desired functionality.

For all these experiments, *the false negative rate is zero.* That is, no consumer-unwanted functionality is available in any of the test binaries after trimming. For example, after instrumenting gcc with training data that uses the -m64 command-line flag to target 64-bit architectures, the trimmed binary is unable to compile any program for 32-bit architectures; specifying -m32 on the command-line yields a security abort. This is because our method is a whitelisting approach that prefers overfitting to maintain high assurance. The same experiments performed using prior CFI-only solutions yield the 100% false negative rate reported in Table 1.

A classification's susceptibility to false positives can be measured in terms of its *false positive ratio* (i.e., the complement of its *specificity*). The false positive ratio of control-flow trimming is driven by the unit testing's ability to comprehensively model the set of semantic features desired by the consumer. We therefore measure accuracy as the false positive ratio, broken down into three measures: the percentage of contexts incorrectly identified as anomalies, the percentage of branch origins that at least one context anomaly incorrectly detected at that branch site, and the total percentage of traces in which at least one anomaly was incorrectly detected.

Table 7 shows the resulting false positive ratios. Each entry in the table is averaged over 10 different experiments in which trace samples are randomly drawn. Since the training phase's accuracy depends significantly on the size of the training data, we conducted experiments with 10–1000 samples for training, evaluation, and testing with a ratio of $3 : 1 : 1$. The experiments consider the effects of two different confidence thresholds for CCFG pruning (see §3.2): 0.0, 0.25, and an optimal threshold t^* experimentally determined as the minimum threshold that achieves zero false negatives for evaluation sample traces. A threshold of 0.0 means no pruning, which is the most conservative CCFI policy (no relaxation). All experiments use contexts of length 4 as described in §4.

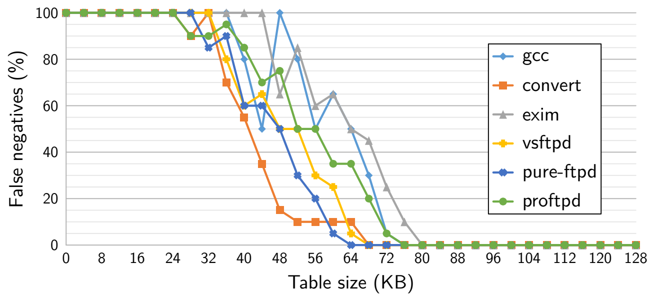

Figure 5: False negative ratios with varying table sizes

As expected, increasing the training data size significantly improves classification accuracy, until at higher training sizes, almost all experiments exhibit perfect accuracy. More aggressive CCFG policy pruning via lower confidence thresholds helps to offset the effects of overfitting when limited training data is available. Increasing context size has a reverse effect; the increased discriminatory power of the classifier (due to widening its feature space by a multiplicative factor for each additional context entry) creates a more complex concept for it to learn. More comprehensive training is therefore typically required to learn the concept.

5.3.2 *Interactive Experiments.* As a whitelisting approach, our design primarily targets software whose desired features are well known to the consumer, and can therefore be fully exercised during training. Table 7 shows that highly interactive products, such as browsers, might require more training to learn all their features as a result of this design. Experiments on epiphany and uzbl require about 500 traces to obtain high accuracy, with a few rare corner cases for epiphany only discovered after about 1000 traces, and uzbl never quite reaching perfect accuracy.

To better understand the relationship between interaction diversity and training burden for such products, Figure 4 plots the accuracy rate for uzbl as the diversity of interactions increases, with the training set size held fixed at 100 traces. Each data point characterizes an experiment in which training and testing are limited to $x \in [1, 12]$ different types of user interactions (e.g., using forward-backward navigation but not page-zoom). The results show an approximately linear decline in accuracy as the diversity of interactions increases, indicating that more training is needed to learn the consumer's more complex policy.

5.3.3 *Table Size.* For efficiency purposes, our enforcement approximates the policy being enforced as a hash table (see §3.3). Poor approximations that use an overly small hash table could permit dangerous false negatives (e.g., undetected attacks), since the enforcement would inadvertently accept policy-violating contexts whose hashes happen to collide with at least one policy-permitted context. To investigate the minimum table sizes needed to avoid these risks, we therefore performed an additional series of experiments wherein we varied the hash table size without changing the policy, and measured the false negative ratio for each size.

Figure 5 plots the results for six of the programs with test suites, with hash table size on the x-axis and false negative ratio on the y-axis. The results show that even hash table sizes as small as 128 bytes (1024 bit-entries) reliably achieve a zero false negative rate. This is because policy-accepted contexts are so rare relative to the space of all possible contexts that almost any sequence of contexts that implements an undesired feature quickly witnesses at least one context that is policy-violating, whereupon it is rejected.

Our experiments nevertheless use larger table sizes than this minimum in order to minimize population ratio p, which §5.2 shows is important for resisting implementation-aware code-reuse attacks. Specifically, table sizes that scale with the code section size are recommended for security-sensitive scenarios where the threat model anticipates that adversaries have read-access to the table, and might use that knowledge to craft gadget chains.

6 DISCUSSION

6.1 Control-flow Obfuscation

Although our evaluation presently only targets non-obfuscated binary code, we conjecture that control-flow trimming via CCFG enforcement has potentially promising applications for hardening obfuscated binaries as well. Instruction-level diversification [21], opaque predicates [47], and control-flow flattening [84] are some examples of code obfuscation and anti-piracy techniques that are commonly applied by code-producers to prevent effective binary reverse-engineering.

For example, flattening adds a dispatcher to the program through which all control-flow transfers are rerouted. This makes it more difficult for adversaries to reverse-engineer the control-flows, but it also prevents context-insensitive CFI protections from securing them, since the flattening transforms individual CFG edges into *chains* of edges that must be permitted or rejected. Context-sensitivity is needed to reject the chain without rejecting the individual edges in the chain. The context-sensitivity of our approach therefore makes it well-suited to such obfuscations.

6.2 Shared Libraries

Our experiments report results for CCFG policies enforced on user-level applications and their dedicated libraries, but not on system shared libraries. Securing system shared libraries can be accomplished similarly, but if the library continues to be shared, its policy must permit all the semantic features of all the applications that import it. This can introduce unavoidable false negatives for the individual applications that share it. We therefore recommend that consumers who prioritize security should avoid shared versions of the system libraries in security-critical applications, so that control-flow trimming can specialize even the system library code to the application's specific usage requirements.

6.3 Concurrency, Non-determinism, and Non-control Data Attacks

Our IRM implementation stores contextual information in thread-local machine registers for safety and efficiency. This immunizes it against context pollution due to concurrency. However, it also means that it cannot block attacks that have no effect upon any thread's control-flow, such as non-control data attacks in which one thread corrupts another thread's data without affecting its own control-flows or those of the victim thread. Such attacks are beyond the scope of all CFI-based defenses [2].

7 RELATED WORK

7.1 Code Surface Reduction

Software debloating has been used in the past to reduce code sizes for performance and security. Such techniques were initially applied to Linux kernels to save memory on embedded systems [19, 35, 45]. Later the focus shifted to reducing the kernel's attack surface to improve security [33, 42–44, 77]. Prior work has shown that certain Linux kernel deployments leave 90% of kernel functions unused [42]. kRazor learns the set of used functions based on runtime traces, and limits the code reachability using a kernel module. Face-Change [33] makes multiple minimized kernels in a VM and exposes each minimized kernel to a particular application upon context-switching. In contrast to these works, our approach is not kernel-specific, can enforce context-sensitive control-flow policies, and can debloat code at instruction-level granularity.

Code surface reduction has recently started to be applied to user-level libraries and programs. Winnowing [48] is a source-aware static analysis and code specialization technique that uses partial evaluation to preserve developer-intended semantics of programs. It implements Occam, which performs both intra-module and inter-module winnowing atop LLVM, and produces specific version of the program based on the deployment setup. Piecewise Debloating [64] uses piece-wise compilation to maintain intra-modular dependencies, and a piece-wise loader that generates an inter-modular dependency graph. The loader removes all code that is not in the dependency graph. Chisel [36] debloats the program given a high-level specification from the user. The specification identifies wanted and unwanted program input/output pairs, and requires the source code and the compilation toolchain. To accelerate program reduction, Chisel uses reinforcement learning. It repeats a trial and error approach to make a more precise Markov Decision Process that corresponds to the specification.

Source-free, binary code reduction has been achieved for certain closed-source Windows applications by removing unimported functions in shared libraries at load time [54]. The approach requires *image freezing*, which prevents any new code section or executable memory page from being added. Shredder [51] is another source-free approach that specializes the API interface available to the application. It combines inter-procedural backwards data flow analysis and lightweight symbolic execution to learn a policy for each function in the program. Although these approaches boast source-freedom, they can only permit or exclude program behaviors at the granularity of functions with well-defined interfaces. Many critical security vulnerabilities, including Shellshock, cannot be isolated to individual functions, so cannot be pruned in this way without removing desired program behaviors. Our approach therefore learns and enforces policies definable as arbitrary CCFGs irrespective of function boundaries or even the feasibility of recovering function abstractions from the binary.

7.2 Control-flow Integrity

SFI [82] and CFI [1] confine software to a whitelist of permitted control-flow edges by guarding control-transfer instructions with dynamic checks that validate their destinations. In SFI, the policy is typically a sandboxing property that isolates the software to a subset of the address space, whereas CFI approaches typically enforce stronger properties that restrict each module's internal flows. In both cases the policy is designed to prohibit flows unintended or unwanted by software developers (e.g., developer-unwanted component interactions or control-flow hijacks). Since the original works, the research community have proposed many variations (e.g., [4, 22, 23, 29, 49, 52, 57–60, 63, 78–80, 85, 89, 91, 92]), most of which improve security, performance, compatibility, and/or applicability to various code domains and architectures.

CFI algorithms come in context-sensitive and context-insensitive varieties. Context-sensitivity elevates the power of the policy language using contextual information, such as return address history or type information, usually in a protected shadow stack. The price of such power is usually lower performance due to maintaining, consulting, and securing the contexts. Low overhead solutions must usually relax policies, introducing a sacrifice of assurance.

For example, kBouncer [61] enforces a context-sensitive policy that considers the previous 16 jump destinations at each system call. Unfortunately, enforcing the policy only at system calls makes the defense susceptible to history-flushing attacks [18], wherein attackers make 16 benign redundant jumps followed by a system call. ROPecker [20] and PathArmor [79] implements OS kernel modules that consult last branch record (LBR) CPU registers to achieve lower performance, which are only available at ring 0. Both systems implement sparse checking regimens to save overhead, in which not every branch is checked. CCFI [49] uses message authentication codes (MACs) to protect important pointers, such as return addresses, function pointers, and vtable pointers, to enforce call-return matching policies.

CFI methodologies can also be partitioned into source-aware and source-agnostic approaches. Source-aware approaches are typically more powerful and more efficient, because they leverage source code information to infer more precise policies and optimize code. However, they are inapplicable to consumers who receive closed-source software in strictly binary form, and who wish to enforce consumer-specific policies. They likewise raise difficulties for software products that link to closed-source library modules. These difficulties have motivated source-agnostic approaches.

WIT [4], MIP [57], MCFI [58], Forward CFI [78], RockJIT [59], CCFI [49], π-CFI [60], VTrust [90], VTable Interleaving [12], PittyPat [26], CFIXX [16], and μCFI[38] are examples of source-aware CFI. XFI [29], Native Client [89], MoCFI [22], CCFIR [91], binCFI [92], O-CFI [52], BinCC [85], Lockdown [63], PathArmor [79], TypeArmor [80], C-FLAT [3], OFI [87], and τCFI [55] are all examples of source-free approaches.

Our research addresses the problem of consumer-side software feature trimming and customization, which calls for a combination of source-agnosticism and context-sensitivity. Binary control-flow trimming is therefore the first work to target this difficult combination for fine-grained CCFG learning and enforcement. Table 1 emphasizes the difference between this problem and the problems targeted by prior works. For example, PathArmor enforces contextual CFG policies, but maintains a much sparser context that is only checked at system API calls. This suffices to block exploitation of developer-unintended features, but not abusable developer-intended functionalities.

7.3 Partial Evaluation

Partial evaluation [40] is a program analysis and transformation that specializes code designed to accommodate many inputs to instead accommodate only a specific subset of possible inputs. This can have the effect of shrinking and optimizing the code, at the expense of deriving code of less generality. Although partial evaluation has traditionally only been applied to source code programs, recent work has applied it to de-bloat native codes without sources. WiPEr [27, 75] lifts Intel IA-32 native code to CodeSurfer/x86 intermediate form [9], converts it to a quantifier-free bit-vector logic amenable to specialization, and then synthesizes specialized native code using McSynth [76]. While the approach is promising, it is currently only applicable to relatively small binary programs with clearly demarcated inputs, such as integers. Larger inputs, such as string command-lines or user-interactive behaviors, prevent the slicing algorithm from effectively extracting and eliminating concept-irrelevant portions of the code automatically.

7.4 Abnormal Behavior Detection

Our approach to learning CCFG policies from traces is a form of anomaly-based intrusion detection, which also has security applications for malware detection and software behavior prediction.

7.4.1 Malware Detection and Code Reuse. Static and dynamic analyses are both used in modern malware detection. Static analysis can be based on source code or binaries, and does not use any runtime information. For example, Apposcopy [30] uses static taint analysis and inter-component call graphs to match applications with malware signatures specified in a high level language that describes semantic characteristics of malware. Static code analysis for malware detection has been proved to be undecidable in general, as witnessed by opaque constants [53], which can obfuscate register-load operations from static analyses. As a result, most of the recent works in this genre use dynamic or hybrid static-dynamic analyses (e.g., [7, 41, 62]). As an example of dynamic analysis, Crowdroid [15] uses system calls, information flow tracking, and network monitoring to detect malware and trojans as they are being executed. TaintDroid [28] is another Android application that constantly monitors the system and detects leaks of user-sensitive information using dynamic taint analysis.

7.4.2 Software Behavior Prediction. Prior works have leveraged machine learning to classify program traces. Markov models trained on execution traces can learn a classifier of program behaviors [13]. Random forests are another effective technique [34]. Software behavioral anomalies have also be identified via intra-component CFGs constructed from templates mined from execution traces [56]. Recent work has also applied clustering of input/output pairs and their amalgamations for this purpose [6]. Our approach adopts a decision tree forest model because of its efficient implementation as in-lined native code (see §4) and its amenability to relaxation and specialization at control-flow transfer points (see §3.2).

8 CONCLUSION

Control-flow trimming is the first work to offer an automated, source-free solution for excluding developer-intended but consumer-unwanted functionalities expressible as CCFGs from binary software products with complex input spaces, such as command-lines, files, user interactivity, or data structures. Using only traces that exercise consumer-desired behaviors, the system learns a contextual CFG policy that whitelists desired semantic features, and inlines an enforcement of that policy in the style of context-sensitive CFI into the target binary. A prototype implementation for Intel x86-64 native code architectures exhibits low runtime overhead (about 1.87%) and high accuracy (zero misclassifications) for training sets as small as 100–500 samples). Experiments on real-world software demonstrate that control-flow trimming can eliminate zero-day vulnerabilities associated with consumer-unwanted features, and resist control-flow hijacking attacks based on code-reuse.

ACKNOWLEDGMENTS

The research reported herein was supported in part by ONR Award N00014-17-1-2995, NSF Award #1513704, and an endowment from the Eugene McDermott family.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proc. 12th ACM Conf. Computer and Communications Security (CCS)*. 340–353.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Information and System Security (TISSEC)* 13, 1 (2009).

[3] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-flow Attestation for Embedded Systems Software. In *Proc. 23rd ACM Conf. Computer and Communications Security (CCS)*. 743–754.

[4] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proc. 29th IEEE Sym. Security & Privacy (S&P)*. 263–277.

[5] Mahmoud Al-Qudsi. 2017. Will AMD's Ryzen Finally Bring SHA Extensions to Intel's CPUs? *NeoSmart Technologies* (2017).

[6] Rafig Almaghairbe and Marc Roper. 2017. Separating Passing and Failing Test Executions by Clustering Anomalies. *Software Quality J.* 25, 3 (2017), 803–840.

[7] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. 2011. Graph-based Malware Detection Using Dynamic Analysis. *J. Computer Virology* 7, 4 (2011), 247–258.

[8] Apache. 2019. Apache benchmark. http://httpd.apache.org/docs/current/programs/ab.html.

[9] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86. In *Proc. 14th Int. Conf. Compiler Construction (CC)*. 250–254.

[10] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proc. 35th IEEE Sym. Security & Privacy (S&P)*. 227–242.

[11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented Programming: A New Class of Code-reuse Attacks. In *Proc. 6th ACM Sym. Information, Computer and Communications Security (ASIACCS)*. 30–40.

[12] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Proc. 23rd Annual Network & Distributed System Security Sym. (NDSS)*.

[13] James F. Bowring, James M. Rehg, and Mary Jean Harrold. 2004. Active Learning for Automatic Classification of Software Behavior. In *Proc. ACM SIGSOFT Int. Sym. Software Testing and Analysis (ISSTA)*. 195–205.

[14] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.

[15] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: Behavior-based Malware Detection System for Android. In *Proc. 32nd IEEE Sym. Security & Privacy (S&P)*. 15–26.

[16] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++. In *Proc. 25th Annual Network & Distributed System Security Sym. (NDSS)*.

[17] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proc. 24th USENIX Security Sym.* 161–176.

[18] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *Proc. 23rd USENIX Security Sym.* 385–399.

[19] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. 2005. System-wide Compaction and Specialization of the Linux Kernel. In *Proc. ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 95–104.

[20] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. 2014. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Proc. 21st Annual Network & Distributed System Security Sym. (NDSS)*.

[21] Frederick B. Cohen. 1993. Operating System Protection Through Program Evolution. *Computer Security* 12, 6 (1993), 565–584.

[22] Lucas Davi, Ra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A Framework to Mitigate Control-flow Attacks on Smartphones. In *Proc. 19th Annual Network & Distributed System Security Sym. (NDSS)*.

[23] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-assisted Flow Integrity Extension. In *Proc. 52nd Annual Design Automation Conf. (DAC)*.

[24] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-in-Time) Return-oriented Programming. In *Proc. 22nd Annual Network & Distributed System Security Sym. (NDSS)*.

[25] Baden Delamore and Ryan K.L. Ko. 2015. A Global, Empirical Analysis of the Shellshock Vulnerability in Web Applications. In *Proc. 1st IEEE Int. Workshop Trustworthy Software Systems (TrustSoft)*.

[26] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-sensitive Control Security. In *Proc. 26th USENIX Security Sym.*

[27] Evan Driscoll and Tom Johnson. 2016. *Lean and Efficient Software: Whole-Program Optimization of Executables.* Technical Report. GrammaTech.

[28] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI)*. 393–407.

[29] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI)*. 75–88.

[30] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Proc. 22nd ACM SIGSOFT Sym. Foundations of Software Engineering (FSE)*. 576–587.

[31] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling Client-side Crash-resistance to Overcome Diversification and Information Hiding. In *Proc. 23rd Annual Network & Distributed System Security Sym. (NDSS)*.

[32] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using Gadget-chain Length to Prevent Code-reuse Attacks is Hard. In *Proc. 23rd USENIX Security Sym.* 417–432.

[33] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2014. Face-Change: Application-driven Dynamic Kernel View Switching in a Virtual Machine. *Proc. 44th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN)*, 491–502.

[34] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. 2005. Applying Classification Techniques to Remotely-collected Program Execution Data. In *Proc. 10th European Software Engineering Conf. (ESEC)*. 146–155.

[35] Haifeng He, Saumya K. Debray, and Gregory R. Andrews. 2007. The Revenge of the Overlay: Automatic Compaction of OS Kernel Code via On-demand Code Loading. In *Proc. 7th ACM/IEEE Int. Conf. Embedded Software (EMSOFT)*. 75–83.

[36] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proc. 25th ACM SIGSAC Conf. Computer and Communications Security (CCS)*. 380–394.

[37] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. 2012. Microgadgets: Size Does Matter in Turing-Complete Return-oriented Programming. In *Proc. 6th USENIX Workshop Offensive Technologies (WOOT)*. 64–76.

[38] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proc. 25th ACM Conf. Computer and Communications Security (CCS)*. 1470–1486.

[39] Intel. 2019. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation, Chapter 2.6.3: Intel Microarchitecture Code Name Nehalem: Execution Engine.

[40] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International.

[41] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. 2009. Effective and Efficient Malware Detection at the End Host. In *Proc. 18th USENIX Security Sym.* 351–366.

[42] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. 2014. Quantifiable Run-Time Kernel Attack Surface Reduction. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 212–234.

[43] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. 2011. Attack Surface Reduction for Commodity OS Kernels: Trimmed Garden Plants May Attract Less Bugs. In *Proc. 4th European Workshop System Security (EUROSEC)*.

[44] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, and Daniel Lohmann. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proc. 20th Network and Distributed System Security Sym. (NDSS)*.

[45] Che-Tai Lee, Zeng-Wei Rong, and Jim-Min Lin. 2003. Linux Kernel Customization for Embedded Systems by using Call Graph Approach. In *Proc. 6th Asia and South Pacific Design Automation Conf. (ASP-DAC)*. 689–692.

[46] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. 26th ACM Conf. Programming Language Design and Implementation (PLDI)*. 190–200.

[47] Anirban Majumdar and Clark Thomborson. 2005. Securing Mobile Agents Control Flow Using Opaque Predicates. In *Proc. 9th Int. Conf. Knowledge-based Intelligent Information and Engineering Systems (KES)*. 1065–1071.

[48] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated Software Winnowing. In *Proc. 30th Annual ACM Sym. Applied Computing (SAC)*. 1504–1511.

[49] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*. 941–951.

[50] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture. In *Proc. 15th USENIX Security Sym.*

[51] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits Through API Specialization. In *Proc. 34th Annual Computer Security Applications Conf. (ACSAC)*. 1–16.

[52] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-flow Integrity. In *Proc. 22nd Annual Network & Distributed System Security Sym. (NDSS)*.

[53] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Proc. 23rd Annual Computer Security Applications Conf. (ACSAC)*. 421–430.

[54] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. Black Hat USA.

[55] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proc. 21st Sym. Research in Attacks, Intrusions, and Defenses (RAID)*. 423–444.

[56] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B. Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly Detection Using Program Control Flow Graph Mining From Execution Logs. In *Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD)*. 215–224.

[57] Ben Niu and Gang Tan. 2013. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proc. 20th ACM Conf. Computer and Communications Security (CCS)*. 199–210.

[58] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *Proc. 35th ACM Conf. Programming Language Design and Implementation (PLDI)*. 577–587.

[59] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-in-Time Compilation Using Modular Control-flow Integrity. In *Proc. 21st ACM Conf. Computer and Communications Security (CCS)*. 1317–1328.

[60] Ben Niu and Gang Tan. 2015. Per-input Control-flow Integrity. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*. 914–926.

[61] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proc. 22nd USENIX Security Sym.* 447–462.

[62] Younghee Park, Douglas S. Reeves, and Mark Stamp. 2013. Deriving Common Malware Behavior Through Graph Clustering. *Computers & Security* 39 (2013), 419–430.

[63] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-grained Control-flow Integrity Through Binary Hardening. In *Proc. 12th Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 144–164.

[64] Anh Quach, Aravind Prakash, and Lok-Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security Sym.* 869–886.

[65] Eric Steven Raymond. 2003. *The Art of Unix Programming.* Addison-Wesley, 313.

[66] Giampaolo Rodola'. 2018. pyftpdlib. https://github.com/giampaolo/pyftpdlib.

[67] Jonathan Salwan. 2018. ROPgadget Tool. https://github.com/JonathanSalwan/ROPgadget. Retrieved 5/6/2018.

[68] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Trans. Information and System Security (TISSEC)* 3, 1 (2000), 30–50.

[69] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming. In *Proc. 36th IEEE Sym. Security & Privacy (S&P)*. 745–762.

[70] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proc. 20th USENIX Security Sym.*

[71] Jeff Seibert, Hamed Okhravi, and Eric Söderström. 2014. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proc. 21st ACM Conf. Computer and Communications Security (CCS)*. 54–65.

[72] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proc. 14th ACM Conf. Computer and Communications Security (CCS)*. 552–561.

[73] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proc. 34th IEEE Sym. Security & Privacy (S&P)*. 574–588.

[74] Solar Designer. 1997. "return-to-libc" attack. *Bugtraq, Aug* (1997).

[75] Venkatesh Srinivasan and Thomas Reps. 2015. Partial Evaluation of Machine Code. In *Proc. 30th ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 860–879.

[76] Venkatesh Srinivasan and Thomas Reps. 2015. Synthesis of Machine Code From Semantics. In *Proc. 36th ACM Conf. Programming Language Design and Implementation (PLDI)*. 596–607.

[77] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Automatic OS Kernel TCB Reduction by Leveraging Compile-time Configurability. In *Proc. 8th Conf. Hot Topics in System Dependability (HotDep)*.

[78] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proc. 23rd USENIX Security Sym.* 941–955.

[79] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-sensitive CFI. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*. 927–940.

[80] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-reuse Attacks at the Binary Level. In *Proc. 37th IEEE Sym. Security & Privacy (S&P)*.

[81] Steven J. Vaughan-Nichols. 2014. Shellshock: Better 'bash' patches now available. *ZDNet* (September 2014). https://www.zdnet.com/article/shellshock-better-bash-patches-now-available.

[82] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proc. 14th ACM Sym. Operating Systems Principles (SOSP)*. 203–216.

[83] James Walden, Jeff Stuckman, and Riccardo Scandariato. 2014. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *Proc. 25th Int. Sym. Software Reliability Engineering (ISSRE)*. 23–33.

[84] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. *Software Tamper Resistance: Obstructing Static Analysis of Programs.* Technical Report. U. Virginia, Charlottesville.

[85] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. 2015. Binary Code Continent: Finer-grained Control Flow Integrity for Stripped Binaries. In *Proc. 31st Annual Computer Security Applications Conf. (ACSAC)*. 331–340.

[86] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *Proc. 24th Conf. USENIX Security Sym.* 627–642.

[87] Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. 2017. Object Flow Integrity. In *Proc. 24th ACM Conf. Computer and Communications Security (CCS)*. 1909–1924.

[88] David A. Wheeler. 2015. Shellshock. In *Learning from Disaster.* https://dwheeler.com/essays/shellshock.html.

[89] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc. 30th IEEE Sym. Security & Privacy (S&P)*. 79–93.

[90] Chao Zhang, Dawn Xiaodong Song, Scott A. Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *Proc. 23rd Annual Network & Distributed System Security Sym. (NDSS)*.

[91] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zo. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proc. 34th IEEE Sym. Security & Privacy (S&P)*. 559–573.

[92] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proc. 22nd USENIX Security Sym.* 337–352.

[93] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *Proc. 3rd Int. Conf. Software Testing, Verification and Validation (ICST)*. 421–428.