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 debolating, control-flow integrity

1 INTRODUCTION

Security of software is widely believed to be inversely related to its complexity (cf., [83, 93]). With more features, larger implementers, and more behavioral variety come more opportunities for program 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
was applied to secure all the binaries against control-flow abuse. To enforce these policies, we therefore introduce a new contextual

...
features due to the modular design of complex software, which reuses 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 \(c_1, c_2, c_3, c_4\). 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 \((c_3, c_5)\) 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 \(c_2, c_3, c_1, c_5, c_4\). This prevents removal of edges \((c_1, c_3)\) and \((c_3, c_5)\) from the CFG, since doing so would break \(F_3\). But an enforcement that permits edge \((c_3, c_5)\) conditional on it being immediately preceded by edge \((c_2, c_1)\) 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 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+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 \(i\)th 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 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 \(\gamma\) and number of contexts \(\lambda\) 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 \(\chi\) we can check whether it is permissible as follows: The last edge in \(\chi\) must be a root of some tree in our learned decision tree forest; otherwise the impending branch is rejected. The penultimate edge in \(\chi\) should be one of that root’s children; otherwise the impending branch is rejected. We continue to check the context edges in \(\chi\) 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 \((e_2, y = 2, \lambda = 5)\), since there are 2 traces and 5 histories having edge \(e_2\). Edge \(e_2\) is the penultimate edge in 4 of those cases, and \(e_1\) is the penultimate edge in 1 case, causing nodes \((e_2, y = 2, \lambda = 4)\) and \((e_1, y = 1, \lambda = 1)\) 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 \(e_1, e_2,\) and \(e_3\) are antepenultimate when \(e_2\) is penultimate, and \(e_3\) 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, \(\lambda\) is the sum of its child \(\lambda\)'s, while \(y\) 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

\[
\text{confidence}(n) = \frac{y}{N} \times \frac{1}{M^2} \sum_{m=1}^{M} \frac{\lambda_m}{\lambda} \log \left( \frac{\lambda_m}{\lambda} \right)
\]

where \((e, y, \lambda)\) is node \(n\)'s label, \(M\) is the number of node \(n\)'s children, \((e_m, y_m, \lambda_m)\) 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 \(\lambda\), 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 \(\lambda\)'s 5 and 5, we have higher confidence than if we observe two children with \(\lambda\)'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 \(r^*\) that maximizes accuracy on the training set is computed using crossfold validation (see §5), and all children with confidence below \(r^*\) 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 \((e_2, y = 2, \lambda = 4)\) are 0.36 and 0.31, respectively. If our confidence threshold exceeds a node’s confidence score, then the tree is disregarded when making policy decisions at that origin. So in our example, a confidence threshold of 0.35 prunes the tree after node \((e_2, y = 2, \lambda = 4)\), 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

\[
\text{hash}(\chi) = \bigoplus_{i=1}^{\lfloor |\chi| \rfloor} ((\pi_2 \chi_i) \ll (|\chi| - i)s)
\]

where \(\bigoplus\) is xor, \(|\chi|\) is the length of context \(\chi\), \(\pi_2 \chi_i\) is the destination (second projection) of the \(i\)th edge in \(\chi\), \(\ll\) 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:

\[
\text{hash}(\chi) = (\text{hash}(\chi) \ll s) \oplus (\pi_2 e)
\]

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 \(s = 1\) and \(w = \log_2 c\) 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.
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.

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.
We experimentally evaluated our control-flow trimming system in we use one centralized table for all interoperating images, accessed via the .got table for concurrent, shared access between modules. For this work, we enforce the policy on all branches from any portion of Table 6, plus bash, gcc, ImageMagic, the epiphany and uzbl browsers, and DynamoRIO, which we use for trace collection and replay.

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

<table>
<thead>
<tr>
<th>Guard Name</th>
<th>Legacy-mode</th>
<th>SHA-extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>before-check</td>
<td>1:movd $r, %xmm12</td>
<td>1:movd $r, %xmm12</td>
</tr>
<tr>
<td></td>
<td>2:psubd %xmm13, %xmm12</td>
<td>2:psubd %xmm13, %xmm12</td>
</tr>
<tr>
<td></td>
<td>3:sha1msg1 %xmm14, %xmm15</td>
<td>3:sha1msg1 %xmm14, %xmm15</td>
</tr>
<tr>
<td></td>
<td>4:sha1msg2 %xmm15, %xmm15</td>
<td>4:sha1msg2 %xmm15, %xmm15</td>
</tr>
<tr>
<td></td>
<td>5:pslldq $4, %xmm15</td>
<td>5:pslldq $4, %xmm15</td>
</tr>
<tr>
<td></td>
<td>6:pxor %xmm12, %xmm15</td>
<td>6:pxor %xmm12, %xmm15</td>
</tr>
<tr>
<td>check</td>
<td>4:movd %xmm15, $r</td>
<td>7:movd %xmm15, $r</td>
</tr>
<tr>
<td></td>
<td>5:and max_hash</td>
<td>8:and max_hash</td>
</tr>
<tr>
<td></td>
<td>6:bt $r, (HASH_TABLE)</td>
<td>9:bt $r, (HASH_TABLE)</td>
</tr>
<tr>
<td></td>
<td>7:jnb TRAP</td>
<td>10:jnb TRAP</td>
</tr>
<tr>
<td>after-check</td>
<td>8:pslrdq $4, %xmm14, %xmm15</td>
<td>11:pslrdq $4, %xmm14</td>
</tr>
<tr>
<td></td>
<td>9:psllidq $4, %xmm14</td>
<td>12:psllidq $4, %xmm14</td>
</tr>
<tr>
<td></td>
<td>10:pxor %xmm12, %xmm14</td>
<td>13:pxor %xmm12, %xmm14</td>
</tr>
<tr>
<td></td>
<td>11:movd $r, %xmm12</td>
<td>14:movd $r, %xmm12</td>
</tr>
<tr>
<td></td>
<td>12:pxor %xmm12, %xmm15</td>
<td>13:pxor %xmm12, %xmm15</td>
</tr>
<tr>
<td></td>
<td>13:psllid $1, %xmm15</td>
<td>14:psllid $1, %xmm15</td>
</tr>
</tbody>
</table>

Figure 3 graphs the runtime overhead for SPEC2017 benchmarks. 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

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.

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 test 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
issue 25,000 requests with concurrency level of 10 for benchmarking lighttpd and nginx. To benchmark the FTP servers, we wrote a script to expose it to a variety of commands (e.g., SITE CPFR and SITE CPTO commands). In node.js we wrote some JS code that 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_start-up configuration variable.

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, ImageMagick 6.8.6–10, ProFTpd 1.3.5, Node.js 8.12, and Exim 4.86.4 that are vulnerable to the CVEs shown in Table 2, including Shellshock and ImageFragick.

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. ImageMagick vulnerabilities afford attackers remote code execution; delete, move, and read access to arbitrary files; and server-side request forgery (SSRF) attack capabilities in ImageMagick 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.js we wrote some JS code that contains a vulnerability. For the remaining programs, we manually exposed each to a variety of inputs representative of common usages. For ImageMagick, 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_start-up 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 6: Space overhead for SPEC2017 intspeed suite benchmarks and some real-world applications

<table>
<thead>
<tr>
<th>Binary</th>
<th>Original Size (KB)</th>
<th>Size Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench_s</td>
<td>10 686</td>
<td>1992</td>
</tr>
<tr>
<td>gcc</td>
<td>63 243</td>
<td>8 499</td>
</tr>
<tr>
<td>mcf_s</td>
<td>13 1</td>
<td>9</td>
</tr>
<tr>
<td>omnetpp_s</td>
<td>28 159</td>
<td>15 67</td>
</tr>
<tr>
<td>cupxalan_s</td>
<td>80 762</td>
<td>47 01</td>
</tr>
<tr>
<td>xz264_s</td>
<td>33 20</td>
<td>567</td>
</tr>
<tr>
<td>deepsjeng_s</td>
<td>50 8</td>
<td>85</td>
</tr>
<tr>
<td>leela_s</td>
<td>3 819</td>
<td>191</td>
</tr>
<tr>
<td>exchange2_s</td>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td>xz_s</td>
<td>1082</td>
<td>146</td>
</tr>
<tr>
<td>exim</td>
<td>1 407</td>
<td>1187</td>
</tr>
<tr>
<td>lighttpd</td>
<td>13 04</td>
<td>294</td>
</tr>
<tr>
<td>memcached</td>
<td>746</td>
<td>156</td>
</tr>
<tr>
<td>nginx</td>
<td>1 674</td>
<td>1444</td>
</tr>
<tr>
<td>openssh</td>
<td>2467</td>
<td>638</td>
</tr>
<tr>
<td>proftpd</td>
<td>3310</td>
<td>803</td>
</tr>
<tr>
<td>pureftpd</td>
<td>470</td>
<td>118</td>
</tr>
<tr>
<td>vsftpd</td>
<td>143</td>
<td>133</td>
</tr>
<tr>
<td>postgresql</td>
<td>757</td>
<td>544</td>
</tr>
<tr>
<td>node_s</td>
<td>36 758</td>
<td>30 059</td>
</tr>
<tr>
<td>median</td>
<td>1541</td>
<td>556</td>
</tr>
</tbody>
</table>

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., int t3). 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 ImageMagick convert is reduced by 94.5%. (The method of computing Table 7 is detailed in §5.3.)
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 849KB 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

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Samples</th>
<th>$t'$</th>
<th>Context Anomalies</th>
<th>Origin Anomalies</th>
<th>Trace Anomalies</th>
<th>Reachable Code (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>t=0.00 t=0.25 t=1t</td>
<td>t=0.00 t=0.25 t=1t</td>
<td>t=0.00 t=0.25 t=1t</td>
<td></td>
</tr>
<tr>
<td>profp</td>
<td>10</td>
<td>0.48</td>
<td>3.04 2.37 1.75</td>
<td>4.51 3.95 2.81</td>
<td>45.00 30.00 25.00</td>
<td>47.31</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.37</td>
<td>0.43 0.17 0.05</td>
<td>1.68 1.02 0.37</td>
<td>3.00 1.50 1.00</td>
<td>47.81</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>47.85</td>
</tr>
<tr>
<td>vsftp</td>
<td>10</td>
<td>0.38</td>
<td>2.45 2.16 1.60</td>
<td>3.74 3.23 1.80</td>
<td>35.00 25.00 25.00</td>
<td>51.11</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.23</td>
<td>0.33 0.07 0.14</td>
<td>0.91 0.17 0.22</td>
<td>2.00 1.50 1.50</td>
<td>51.47</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>51.47</td>
</tr>
<tr>
<td>pure-ftpd</td>
<td>10</td>
<td>0.41</td>
<td>2.23 1.96 1.43</td>
<td>3.61 3.14 2.83</td>
<td>25.00 25.00 10.00</td>
<td>49.89</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.28</td>
<td>0.04 0.00 0.00</td>
<td>0.15 0.00 0.00</td>
<td>2.50 1.50 1.50</td>
<td>50.63</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>50.63</td>
</tr>
<tr>
<td>exim</td>
<td>10</td>
<td>0.25</td>
<td>2.72 1.12 1.88</td>
<td>5.12 4.06 4.81</td>
<td>35.00 15.00 20.00</td>
<td>10.31</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.53</td>
<td>0.58 0.01 0.00</td>
<td>1.36 0.01 0.00</td>
<td>7.50 1.00 0.00</td>
<td>10.63</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>10.65</td>
</tr>
<tr>
<td>ImageMagic</td>
<td>10</td>
<td>0.64</td>
<td>0.21 0.10 0.04</td>
<td>1.51 1.23 0.91</td>
<td>20.00 15.00 10.00</td>
<td>5.27</td>
</tr>
<tr>
<td>convert</td>
<td>100</td>
<td>0.54</td>
<td>0.09 0.07 0.00</td>
<td>0.17 0.10 0.00</td>
<td>2.50 1.00 0.00</td>
<td>5.53</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>5.55</td>
</tr>
<tr>
<td>gcc</td>
<td>10</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>7.66</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>7.66</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>7.66</td>
</tr>
<tr>
<td>epiphany</td>
<td>10</td>
<td>0.93</td>
<td>10.91 0.22 0.00</td>
<td>19.60 1.29 0.00</td>
<td>85.00 40.00 0.00</td>
<td>23.41</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.81</td>
<td>10.76 0.20 0.08</td>
<td>15.50 1.14 0.57</td>
<td>40.00 10.00 6.50</td>
<td>23.73</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.33</td>
<td>2.94 0.01 0.01</td>
<td>12.14 0.09 0.08</td>
<td>8.70 4.00 0.30</td>
<td>24.01</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>0.00 0.00 0.00</td>
<td>24.01</td>
</tr>
<tr>
<td>uzbl</td>
<td>10</td>
<td>0.92</td>
<td>2.16 0.25 0.12</td>
<td>18.90 1.30 0.81</td>
<td>90.00 40.00 30.00</td>
<td>30.81</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.83</td>
<td>2.09 0.04 0.03</td>
<td>17.56 0.96 0.75</td>
<td>50.50 3.50 2.50</td>
<td>30.83</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.65</td>
<td>0.57 0.01 0.01</td>
<td>9.08 0.34 0.17</td>
<td>10.70 0.50 0.60</td>
<td>30.91</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.45</td>
<td>0.46 0.03 0.02</td>
<td>7.94 0.52 0.33</td>
<td>4.30 0.85 0.35</td>
<td>30.91</td>
</tr>
</tbody>
</table>
which is the most conservative CCFI policy (no relaxation). All experiments use contexts of length 4:1:1. 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 10–100 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.

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.

5.3.3 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-insensitivity 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-free, 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., TypeArmor, 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.

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. CCPI [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], MCFL [58], Forward CFI [78], RockJIT [59], CCFI [49], π-CFI [60], VTrust [90], VTable Interleaving [12], PTTY-Par [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 cCFI [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 de-bloat native codes without sources. WiPEx [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 McSYNTTH [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 been 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
