Hurdle: Securing Jump Instructions Against Code Reuse Attacks

Christian DeLozier
delozier@usna.edu
United States Naval Academy

Kavya Lakshminarayanan
kavyal@seas.upenn.edu
University of Pennsylvania

Gilles Pokam
gilles.a.pokam@intel.com
Intel Corporation

Joseph Devietti
devietti@cis.upenn.edu
University of Pennsylvania

Abstract

Code-reuse attacks represent the state-of-the-art in exploiting memory safety vulnerabilities. Control-flow integrity techniques offer a promising direction for preventing code-reuse attacks, but these attacks are resilient against imprecise and heuristic-based detection and prevention mechanisms.

In this work, we propose a new context-sensitive control-flow integrity system (Hurdle) that guarantees pairwise gadgets cannot be chained in a code-reuse attack. Hurdle improves upon prior techniques by using SMT constraint solving to ensure that indirect control flow transfers cannot be maliciously redirected to execute gadget chains. At the same time, Hurdle's security policy is flexible enough that benign executions are only rarely mischaracterized as malicious. When such mischaracterizations occur, Hurdle can generalize its constraint solving to avoid these mischaracterizations at low marginal cost.

We propose architecture extensions for Hurdle which consist of an extended branch history register and new instructions. Thanks to its hardware support, Hurdle enforces a context-sensitive control-flow integrity policy with 1.02% average runtime overhead.

• **Security and privacy → Hardware security implementation**; *Malware and its mitigation.*

control-flow integrity; code-reuse attacks; SMT solvers

ACM Reference Format:
Christian DeLozier, Kavya Lakshminarayanan, Gilles Pokam, and Joseph Devietti. 2020. Hurdle: Securing Jump Instructions Against Code Reuse Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378506

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378506

1 Introduction

In the arms race of memory safety exploits and defenses, code-reuse attacks have emerged as a way to circumvent recent defenses like W ⊕ X page protections that ensure that executable pages are read-only. Instead of writing exploit code to memory, code-reuse attacks use existing snippets of code, known as *gadgets*, to execute an attack. Code-reuse attacks have been resistant to a variety of proposed prevention mechanisms, including Address Space Layout Randomization (ASLR) [1] and Control Flow Integrity (CFI) techniques [2–5]. Hardware support for security is a critical component of an effective defense, as hardware-based defenses can provide high compatibility and performance.

In order to execute a code-reuse attack, an adversary must be able to exploit one or more memory safety vulnerabilities, redirect the control-flow of the running application, and find sufficient useful gadgets. Memory safety techniques can prevent an attacker from using a memory safety vulnerability to launch an attack, but these techniques are often invasive and expensive [6–8]. Using a *shadow stack* [9, 10] can prevent an attacker from redirecting control-flow using return instructions, but preventing an attacker from abusing indirect jumps and calls is more difficult.

Control-flow integrity techniques [9, 11–23] aim to prevent code-reuse attacks by restricting the execution of an application to the control-flow graph statically defined by the application's code. Traditional CFI (TCFI) systems verify the targets of indirect control-flow transfers. The targets of indirect transfers are verified against a pre-computed static control-flow graph to ensure the target is valid. Consider the code in Figure 1. A TCFI scheme would ensure the indirect function call in basic block E targets X or Y, but nothing else. In practice, many TCFI systems rely on heuristics to lower performance costs. Due to these sources of imprecision, savvy attackers have designed code-reuse attacks to avoid detection under many TCFI schemes [2–5].

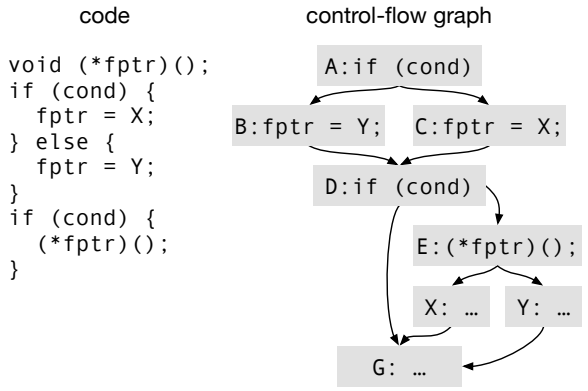

Figure 1. Example code illustrating the security properties of different CFI schemes.

Context-sensitive CFI (CSCFI) schemes [24–26] generalize TCFI by verifying a *path* of control-flow edges *preceding and including* the upcoming edge. This restricts an attacker's capabilities to a greater extent than TCFI systems do, because it verifies not just that, *e.g.*, an indirect function call c goes to a valid target but also that execution has reached c via a valid series of CFG edges by verifying the last k edges. The static CFG is again used to define the set of valid CFG edges. Considering the code in Figure 1 again, and validating the last two CFG edges, CSCFI would allow the indirect function call at E to target X or Y only if execution had reached D via either B or C. Thus, only the four control-flow paths *ABDEX*, *ABDEY*, *ACDEX* and *ACDEY* are permitted, whereas a TCFI scheme would permit any path with a suffix of *EX* or *EY*.

Our proposed scheme, Hurdle, provides tighter security than even CSCFI schemes by performing checks at *every* indirect control-flow transfer (prior CSCFI schemes [24–26] only validate the path at sensitive system calls), and furthermore by inferring the context for each CFG edge from a program's dynamic behavior instead of relying on the static CFG which, as we demonstrate, can be very conservative. Given the code in Figure 1, Hurdle will permit just the path *ACDEY* because it can infer the correlation between the if condition and the call target.

Figure 2 gives an overview of the Hurdle system. Hurdle builds on top of a hardware shadow stack mechanism such as Intel's forthcoming Control Enforcement Technology (CET) [27], which secures return instructions. Hurdle protects indirect jumps and calls from attacker manipulation as well. First, Hurdle observes multiple runs and inputs of a program to infer the context for each indirect jump and call. Second, Hurdle feeds this training data to an SMT solver [28] to construct a set of patterns that are 1) consistent with the program's observed behavior and 2) guaranteed by construction to prevent gadget chaining. While the solver may fail to find a set of acceptable patterns, we find that in practice Hurdle scales to large programs like gcc, with over a million lines of

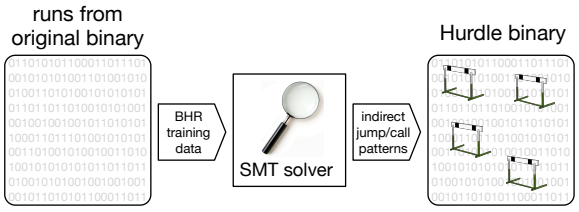

Figure 2. An overview of the Hurdle system.

code. Finally, Hurdle inserts a new instruction before each indirect jump and call that performs a hardware-based runtime check. At runtime, this check compares the program's current control flow history with the pre-computed pattern and flags any violations as potential code reuse attacks. Spurious violations that may arise due to incomplete training data are resolved by stitching pre-computed static paths into Hurdle's dynamic constraints.

This paper makes the following contributions:

- A context-sensitive CFI system that prohibits all pairwise gadget chaining by formulating security as SMT formulae
- A method for pre-computing static control-flow patterns that can augment training data to avoid *all* false positives in our workloads
- Lightweight architecture support for tracking control-flow histories and checking them at runtime
- Simulation of Hurdle's hardware support on a range of embedded and desktop workloads
- A demonstration that Hurdle induces only mild increases in binary size (2.5 KB on average) and performance overhead (1.02% on average)

The remainder of this paper is organized as follows. Section 2 provides background on code-reuse attacks. Section 3 describes the threat model used in the design of this system. Section 4 describes Hurdle, and Section 5 describes Hurdle's hardware support. Section 6 discusses our experimental setup, and Section 7 our evaluation of Hurdle's security and performance. Section 8 discusses related work.

2 Background and Motivation

Indirect control-flow transfers (returns, indirect calls and indirect jumps) are used to implement common programming language constructs such as switch statements and virtual functions. The following sections provide background on code-reuse attacks and defenses against them.

2.1 Code-Reuse Attacks

Conventional code-reuse attacks exploit memory safety vulnerabilities, such as buffer overflows and use-after-free errors, to redirect control flow to malicious code. A code-reuse

attack has a few pre-requisites. The application must have at least one vulnerability that can be exploited by an attacker to overwrite memory. Preventing memory safety vulnerabilities can prevent code-reuse attacks [6, 29] but entails high performance overheads. Given a memory safety exploit, an attacker must also be able to redirect the control flow of the application. Indirect control-flow transfers are particularly useful as they allow arbitrary control-flow redirection. Prior approaches have attempted to limit the possible targets of these indirect transfers [27]. Finally, an attacker must have access to a sufficient number of gadgets to perform a call to a sensitive system call, such as `mmap` or `mprotect`. Once the attacker has control of page permissions, they can disable W ⊕ X protections and execute arbitrary injected code.

2.2 Shadow Stacks

A *shadow stack* prevents return-oriented programming (ROP) attacks [9, 10] by forcing each return instruction to return to its matching call site. At each call instruction, the shadow stack records the PC to return to once the callee is finished. Shadow stack designs are often implemented in hardware both to avoid non-trivial runtime overheads and to prevent attackers from compromising the integrity of shadow stack contents. Figure 4 shows the reduction in available gadgets when a shadow stack is employed. This initial study contains a subset of the benchmarks evaluated in later portions of the paper due to space constraints. We use reduction of gadgets as a metric to compare prior work because these approaches do not completely eliminate code reuse attacks but rather attempt to make them more difficult by reducing the available attack survace. The leftmost bars (*Conventional*) show a lower bound on the number of gadget pairs available in an unsecured system. This is a lower bound because we do not account for gadgets that could be manufactured by jumping into the middle of an instruction to find bits that look like, say, an indirect jump. This is a limitation of our methodology for generating the figure. In reality, attackers can identify more gadgets by using non-standard offsets (*i.e.* finding a return instruction in an immediate). The *Shadow Stack* bars show the reduced number of gadget pairs available (about 4x fewer) when a shadow stack is employed. Nevertheless, many gadgets remain even with a shadow stack.

A shadow stack prevents code-reuse attacks based on return-oriented programming, but it does not prevent call-oriented programming (COP) or jump-oriented programming (JOP) attacks [30, 31]. These attacks do not use return instructions to hijack control flow from the application and therefore pass all shadow stack checks.

2.3 Intel CET

Intel's Control Enforcement Technology (CET) [27] will provide a hardware CFI scheme in upcoming processors. CET includes a hardware shadow stack and provides a new EndBranch instruction that limits the power of JOP and COP attacks. Each indirect jump or call must target an EndBranch instruction. Any indirect jumps or calls that do not land at an EndBranch instruction raise a hardware exception.

```
ENDBRANCH
test rbp, rbp
jz 0x4683b6
mov rsi, r14
mov edi, r15d
call [r12]
```

```
ENDBRANCH
test eax, eax
mov ebx, eax
jnz 0x7fbbd56a802c
mov r9, [rsp+0x30]
call r9
```

Figure 3. Two EndBranch gadgets found in `bitmnp`. Assuming attacker control of registers and memory, the gadgets can be chained to call `mprotect` with arbitrary input parameters.

Although CET represents a step forward in available control-flow integrity tools, CET does not prevent carefully crafted code-reuse attacks that target "EndBranch gadgets." An EndBranch gadget is a gadget that begins with an EndBranch instruction and ends with an indirect jump or call. Even with CET, such gadgets can be targeted by any indirect jump or call, because CET does not constrain a particular jump or call to land at any particular EndBranch instruction(s).

Figure 3 lists two gadgets found in the `bitmnp` program (from the Autobench suite [32]) that can be used to call `mprotect` with arbitrary input values. Calling these gadgets from an indirect jump or call will not trigger CET's prevention mechanism because they both begin with an EndBranch instruction; the EndBranches are there because `bitmnp` has indirect calls or jumps that, in normal execution, target these locations. We assume that the address of the second gadget is preloaded into the address pointed to by `r12`. The first gadget can be called from any indirect jump or call in the application. Thus, if the attacker can find a memory safety vulnerability and an indirect branch that can be redirected to the first gadget, the attacker can compromise W ⊕ X protections.

As Figure 4 shows, EndBranch gadgets are even rarer (by about 81x) than those permitted by a shadow stack but there are still EndBranch gadgets in all of our benchmark programs. We make no assumption about the length of EndBranch gadgets, but we do assume that EndBranch gadgets contain at most one direct branch. While CET does reduce the number of gadgets beyond what a simple shadow stack permits, there are still thousands of EndBranch gadgets per application for creative attackers to exploit.

2.4 Context-Sensitive CFI

CSCFI schemes use the static CFG to validate a portion of the execution history before vulnerable control-flow transfers. To quantify the conservatism inherent in this approach, we calculated how many distinct static paths of length k

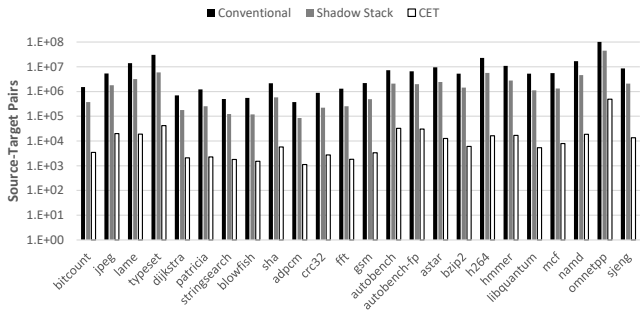

Figure 4. Usable gadget pairs under various protection schemes.

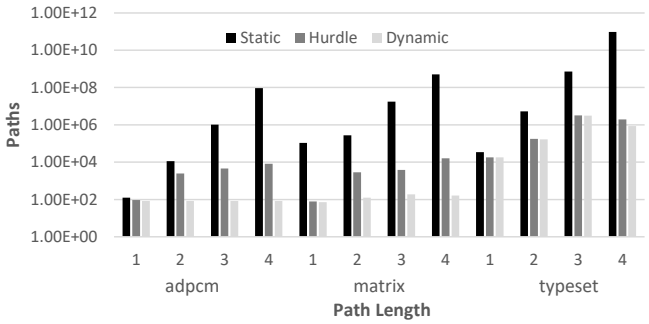

Figure 5. Feasible control-flow paths to each indirect jump/call, for increasing path length in three of our workloads with the smallest, average, and largest number of paths. Note the log y-axis.

terminate at an indirect jump/call[1]. We re-implemented the PathArmor [24, 25] static CFG analysis in Pin [33] and extended it to count static paths. For example, for the CFG in Figure 1, there are two paths (*ABD* and *ACD*) of length two that reach the indirect call in *D*. Intuitively, the more paths there are, the easier attacks are to find.

The *Static* bars in Figure 5 show results for our embedded workloads with the smallest (adpcm), average (matrix) and largest (typeset) number of paths. The number of static paths grows exponentially in the path length for all workloads (note the log y-axis). While Hurdle operates on paths of length 8, we were not able to explore static paths longer than 4 as typeset already had nearly 100B paths. The *Dynamic* bars in Figure 5 show the number of static paths that are consistent with some dynamic control-flow path encountered while running the application. These bars represent a lower bound for how many paths Hurdle might allow in the best case, ignoring security and the benefits of generalizing beyond specific dynamic executions. The *Dynamic* bars are much smaller than the set of possible paths, and do not grow at nearly the same rate.

Overall, our path enumeration experiments show that, even with CSCFI, many opportunities for code-reuse attacks remain. In particular, there are exponentially many permitted paths through the code, normally unused by the application, that are potentially exploitable by a clever attacker.

2.5 Hurdle

Given the large gap between the *Static* and *Dynamic* bars in Figure 5, we would like a CFI scheme that could hew more closely to the application's normal execution patterns. Hurdle improves on CSCFI by learning an application's dynamic execution patterns. In Figure 5, the Hurdle bars show the number of paths allowed by our approach, which is relatively close to the lower bound of dynamic paths, and which does not grow exponentially as the static paths do.

[1]While [24, 25] only validates execution history before dangerous system calls, we investigate the behavior of a stronger version that performs checks at every indirect jump/call like Hurdle does.

3 Threat Model

We assume that $W \oplus X$ protections prevent modifications to existing code and insertion of new code. We assume that an attacker can arbitrarily modify memory through the exploitation of a memory safety error in the application. We assume that the attacker must chain at least two gadgets together in order to execute a code-reuse attack. We assume that the attacker must use gadgets to gain control of values stored in registers. We also assume that Intel CET has been applied to the application, providing both a shadow stack and EndBranch instructions. Given this threat model, Hurdle aims to prevent *EndBranch gadgets* (Section 2.3) from being chained together. Hurdle relies on monitoring the dynamic control-flow of the application, and therefore Hurdle does not prevent data-centric attacks that do not affect control-flow. Hurdle does not handle just-in-time (JIT) compilation or dynamically-linked libraries due to the need to map histories of branch behavior to patterns used to secure future behavior. We leave these extensions to future work.

4 Hurdle

Hurdle prevents code-reuse attacks by requiring that the control-flow history exhibited by the application matches a known pattern at each indirect jump/call. By requiring the current control flow to match a known pattern, Hurdle allows safe executions and prevents malicious executions. If an attacker attempts to perform a code-reuse attack on an application protected by Hurdle, the control flow is unlikely to match the expected control flow. Hurdle aims to only allow the control-flow paths that a normal dynamic execution of the application follows.

4.1 Overview

Hurdle hardens executables against code-reuse attacks in three steps. First, Hurdle **records** the dynamic control-flow histories for a set of training inputs. Second, Hurdle uses

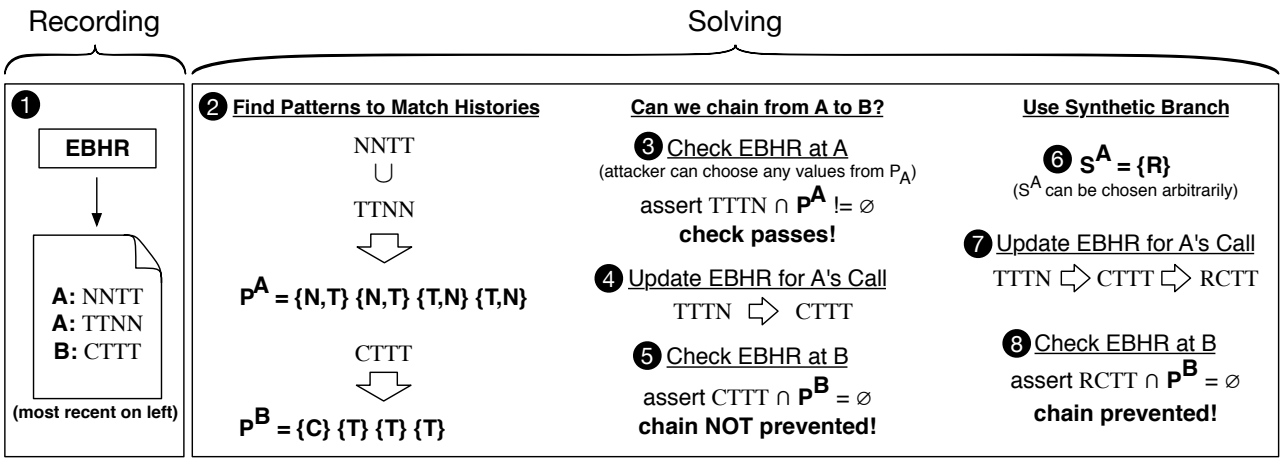

Figure 6. Hurdle in operation. Hurdle collects EBHR histories and combines those histories to form a gate pattern. Hurdle further constrains the gate patterns to prevent gadget chains, which are used in code-reuse attacks. If two patterns are too similar to prevent chains naturally, Hurdle inserts a synthetic branch to manufacture the necessary mismatch.

SMT **solving** to compute the gate patterns for each indirect branch such that the pattern matches all known dynamic control-flow histories and prevents chaining between gadgets. Once the patterns have been computed, Hurdle **instruments** the binary with instructions that check the dynamic control-flow history of the running application against the computed gate pattern. If a pattern check fails at runtime, the control flow of the running application does not match the expected control flow, and the application programmer can specify whether to immediately abort the execution or tolerate some number of failures. When a new control-flow history is discovered, perhaps by running additional inputs, Hurdle can **incrementally update** the existing patterns to allow the new history. Figure 6 shows a detailed example of how Hurdle computes and refines patterns for indirect branches to prevent code-reuse attacks. We work through this example in the sections below.

4.2 Recording Histories

Hurdle uses an extended branch history register (EBHR) to record and check dynamic control-flow histories. The EBHR records a larger set of control-flow events than the existing BHR, which only records taken and not-taken events. The EBHR categorizes control-flow events using the following "alphabet": taken direct branches (T), not-taken direct branches (N), unconditional direct branches (U), direct calls (K), indirect calls (C), indirect jumps (J), and returns (R). The EBHR provides more precise information about the dynamic control-flow history than a conventional BHR, which allows Hurdle to better prevent code-reuse attacks.

A *dynamic control-flow history* is a snapshot of the EBHR at some time during the application's execution. A snapshot is taken immediately preceding each indirect jump/call. Each dynamic history is denoted as D^n where $n \leq N_X$ and N_X is the total number of dynamic histories recorded for indirect jump/call X. Each element D_i^n of the dynamic control-flow history is taken from the set of control-flow events recorded in the EBHR ($\{T, N, U, K, C, J, R\}$).

In Figure 6, the leftmost panel (❶) shows three histories recorded from a 4-entry EBHR. Two are from an indirect call with PC A, and one with PC B.

Our history recording infrastructure is implemented as a Pintool [33]. At each indirect jump/call, Hurdle records the current program counter, the current value of the EBHR, and the program counters of any indirect jumps/calls that are present in the current EBHR. The current PC is needed so that we can associate this history with the appropriate static indirect jump/call. The PCs of other indirect jumps/calls are needed to ensure that Hurdle does not disallow control flow that the application legitimately performs, even though it may look like "gadget chaining" due to the close proximity of indirect jumps/calls. Hurdle only needs to record each unique control-flow history, which helps keep the space costs of recording manageable. The PCs of other indirect jumps/calls are only used during recording and not during production. Thus, the PCs of other indirect jumps/calls are not stored in the EBHR hardware structure and are simply saved in software during the recording process.

4.3 Solving for Gate Patterns

At each indirect jump/call, Hurdle inserts an instruction that checks the dynamic control-flow history against a *gate pattern*, P. Each element P_i of the gate pattern is a non-empty subset of the EBHR alphabet $\{T, N, U, K, C, J, R\}$. For example, a pattern element might be $\{T, N\}$. A dynamic control-flow history *matches* a gate pattern if $\forall i\ D_i^n \in P_i$.

For example, a two-entry pattern $P = \{T, N\}\{J, C\}$ matches the dynamic history TJ, but does not match the history JJ because the first history element $J \notin \{T, N\}$.

Once Hurdle has recorded control-flow histories for a representative set of inputs to the application, it uses the control-flow histories to compute a *gate pattern* for each static indirect jump/call. The gate pattern must satisfy two key requirements: 1) all known control-flow histories must match the gate pattern, and 2) each indirect jump/call X must be unable to chain to another indirect jump/call Y unless explicitly required by the application. Because these requirements (particularly the non-chaining requirement) are non-trivial to determine, Hurdle formulates these requirements as constraints and uses an SMT solver to compute the gate patterns for each indirect jump/call.

In the following sections, we will use the following dynamic histories for indirect calls A and B as an example. Our examples use a simplified EBHR alphabet of $\{N, T, C\}$. These histories are also shown in Figure 6 (❶).

$$D_0^A = NNTT$$
$$D_1^A = TTNN$$
$$D_0^B = CTTT$$

Each gate pattern must match all known dynamic histories (❷). A gate pattern element, P_i, matches one entry of dynamic history, D_i^n, if $D_i^n \in P_i$. At runtime, a pattern must match all entries of the dynamic EBHR for the check to pass, so a pattern P must match all entries of dynamic history D. For performance, these constraints can be simplified to a single assignment for each pattern element, $P_i = \bigcup_{n=1}^{N} D_i^n$ where N is the total number of histories recorded for the indirect jump/call.

For the example indirect calls, we produce the following patterns based on the dynamic histories:

$$P^A = \{N, T\}\{N, T\}\{N, T\}\{N, T\}$$
$$P^B = \{C\}\{T\}\{T\}\{T\}$$

To prevent gadget chains, we assume that each indirect jump/call may be used as an EndBranch gadget and we examine how the histories permitted by one gadget may match (or not) with another gadget.

As an example, we examine chaining from the gadget terminated by A to that terminated at B. We refer to these as gadgets G_A and G_B, respectively. Conceptually, we first examine all of the histories that are permitted by the gate pattern P^A. $TTTN$ is one such history (❸). Assuming we pass the Hurdle runtime check, the indirect call A will get executed, prepending a C to the EBHR and shifting out the oldest entry making it $CTTT$ (❹). Assuming the attacker has somehow manipulated indirect call A to target the start of G_B, the EBHR will contain $CTTT$ when B is reached. Next we examine whether this EBHR history matches P^B, which in this case it does (❺). Thus, our patterns P^A and P^B have failed to prevent gadget chaining, and a sufficiently clever attacker who manufactures the EBHR contents of $TTTN$ can invoke G_A and then G_B in succession. While the practical damage caused by just two gadgets is perhaps small in practice, we seek a strong security guarantee that no such chaining is possible, avoiding assumptions about gadget chain length.

We will show how to repair our example and prevent chaining below, but first we discuss how to express anti-chaining constraints to the solver. Exhaustively examining all possible histories for each pattern and how they interact with each other gadget is too slow, so instead we use logical constraints to guarantee the absence of gadget chaining. Our SMT solver uses these constraints when finding patterns for each indirect jump/call so that, if it can find satisfying patterns, they prevent chaining by construction. Returning to our example in Figure 6, the following constraint expresses our desire to prevent chaining from G_A to G_B:

$$\begin{aligned} NoChain_{A\to B} &= (\{C\} \cap P_0^B = \emptyset) \vee (P_0^A \cap P_1^B = \emptyset) \\ &\quad \vee (P_1^A \cap P_2^B = \emptyset) \vee (P_2^A \cap P_3^B = \emptyset) \\ \\ &= (\{C\} \cap \{C\} = \emptyset) \vee (\{N, T\} \cap \{T\} = \emptyset) \\ &\quad \vee (\{N, T\} \cap \{T\} = \emptyset) \vee (\{N, T\} \cap \{T\} = \emptyset) \end{aligned}$$

The top formula is symbolic, and the bottom version substitutes the concrete values from our example. First we note that, in order to pass the runtime check just before A, each EBHR entry must have matched the corresponding entry in P^A. *After* the indirect call A, the EBHR has the value $C\alpha\beta\gamma$, where C represents the indirect call A (the most recent entry) and $\alpha \in P_0^A$, $\beta \in P_1^A$ and $\gamma \in P_2^A$ represent the fact that the older EBHR entries must have matched with P^A. The entry that matched with P_3^A has been shifted out of the EBHR at this point.

Taking the top formula one piece at a time, $(\{C\} \cap P_0^B = \emptyset)$ represents our desire that the indirect call A (which puts a C into the most-recent EBHR entry) should mismatch with the first element of P^B. $(P_0^A \cap P_1^B = \emptyset)$ and the other similar clauses serve a similar purpose, but we have to align P^A and P^B appropriately to take into account the presence of the latest indirect call. We use a disjunction to join the clauses together because we don't need mismatches in every entry of P^B, but we do need at least one. If there are no mismatches, then the history an attacker used to get past P^A will permit them to chain directly to G_B and P^B will allow them through again. As the concrete values on the bottom show, this anti-chaining constraint is not satisfiable so chaining is indeed possible. Given the need to match the dynamic histories for A and B, there will in fact be a match in every entry in P^B.

When generating anti-chaining constraints, we make exceptions for "gadget pairs" that are required by the program and omit the anti-chaining constraints for those pairs. We

also generate "reflexive" constraints to ensure that a gadget cannot chain to itself. As mentioned in Section 2.3, gadgets may contain a direct branch. If some gadget G_D contains a direct branch, we account for this by offsetting the pattern elements by one when generating anti-chaining constraints. Similarly, when examining chaining from G_D to some other gadget, we account for the direct branch that precedes G_D's indirect call/jump.

To force a pattern mismatch and prevent gadget chaining, we introduce a new mechanism of synthetic branches, explained next.

4.3.1 Synthetic Branches

To prevent gadget chaining when appropriate pattern mismatches do not arise naturally, we insert a *synthetic branch* S^X into the EBHR after an indirect jump/call X. In our running example, we insert a synthetic branch to force the history after executing G_A to mismatch with P^B. The SMT solver has more flexibility to choose the value for S^A because it is unconstrained by any dynamic histories. Using a synthetic branch S^A for indirect call A, our anti-chaining constraint becomes:

$$\begin{aligned} NoChain_{A\to B} &= (S_0^A \cap P_0^B = \emptyset) \vee (\{C\} \cap P_1^B = \emptyset) \\ &\quad \vee (P_0^A \cap P_2^B = \emptyset) \vee (P_1^A \cap P_3^B = \emptyset) \\ \\ &= (\{R\} \cap \{C\} = \emptyset) \vee (\{C\} \cap \{T\} = \emptyset) \\ &\quad \vee (\{N,T\} \cap \{T\} = \emptyset) \vee (\{N,T\} \cap \{T\} = \emptyset) \end{aligned}$$

Supposing our solver chose $S^A = R$ for a synthetic return (❻), the EBHR update as a result of G_A is now different and includes the synthetic branch (R) after the indirect call C (❼). By changing the EBHR when we reach B, the chaining from G_A to G_B is prevented (❽). To prevent the synthetic branch from spuriously conflicting with valid histories, we also need to adjust all known dynamic histories to account for the insertion of synthetic branches, which affects pattern generation in turn. For all pattern constraints that contain an indirect call or jump, we insert the proper synthetic branch afterwards when necessary.

4.3.2 Updating Patterns with Static Pattern Stitching

Hurdle must be trained on dynamic histories that are representative of the control-flow behavior of an application in order to avoid *false positives* – flagging normal execution as a code-reuse attack. False positives can occur in two distinct cases. If the training input does not provide enough code coverage to execute all indirect jumps and calls, there may be no known dynamic histories for a particular indirect jump or call. In this case, false positives can be mitigated by assigning a gate pattern that prohibits indirect jumps and calls ($\{T, N, U, K, R\}$). This pattern trivially satisfies the *NoChain* constraints and may avoid false positives in the event that the training input is not representative of an actual input to the application. In the second case, a known indirect jump or call may be reached by a control-flow path not covered by the training input.

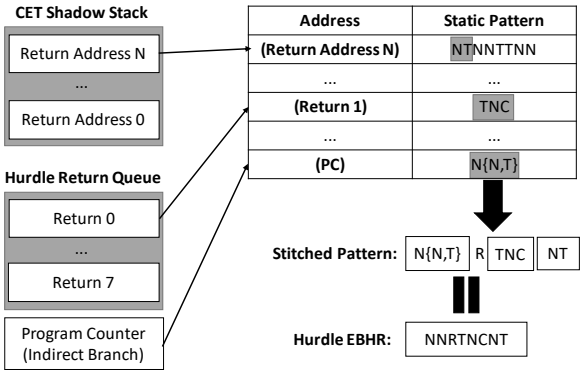

Figure 7. Stitching static patterns using dynamic return addresses to match against dynamic control-flow histories

To determine whether this latter case is part of a code-reuse attack, Hurdle uses *static pattern stitching*. Hurdle computes a *static gate pattern* of the static control-flow paths that reach each indirect call, indirect jump, and return instruction. Hurdle builds a control-flow graph of the binary and walks backward from each indirect call, jump, and return, stopping the traversal when it reaches either the start of the containing function or a path of length 8. Static gate patterns compress all static paths much like a Hurdle gate pattern compresses previously-seen dynamic paths.

When Hurdle encounters a pattern mismatch for a known indirect jump or call, it checks the static gate pattern to determine if it matches the current EBHR; if the static gate pattern does not match then a violation is flagged. Figure 7 illustrates how Hurdle leverages the CET shadow stack and an additional return queue to stitch *multiple* static patterns together to check the full EBHR history. By using this strategy, Hurdle can offer meaningful constraints on the behavior of indirect branches even along paths not seen during training. Static pattern stitching is performed at runtime and does not require extra training or SMT solving.

In the offline setting, Hurdle can also incrementally refine gate patterns to improve accuracy for future runs. When updating a single pattern, all or most other patterns can be initially set as hard-coded constraints that match the previously-computed solution, which accelerates SMT solving time. Section 7 discusses the running time of updating patterns and characterizes the update process.

5 Hurdle Architecture Support

Once Hurdle has identified suitable patterns for each indirect jump/call, this information must be made available at runtime for Hurdle's hardware checks. We discuss here how patterns are encoded into a program binary, and the hardware support required for checking.

5.1 Encoding patterns

Patterns are encoded directly into the immediate part of a new Hcheck instruction. The immediate encodes the gate pattern that must be checked before the indirect jump or call can be retired. The immediate encodes the gate pattern as eight 7-bit gate pattern elements, which requires 7 bytes. The Hcheck instruction also encodes a 7-bit synthetic branch.

5.2 Extended BHR

The next key piece of Hurdle state is the extended BHR (EBHR). EBHR entries are 7 bits in size, using a 1-hot encoding to facilitate checking.[2] As with the conventional BHR, EBHR entries are shifted when a new branch entry is inserted so that the most recent k entries are retained. To avoid adding extra complexity to latency-critical branch prediction operations, we propose maintaining the EBHR in addition to the conventional BHR. A new EBHR entry is inserted at decode, since by then we know the branch type (for UKCJR entries) and its predicted direction (for TN entries). Note that EBHR entries may correspond to speculative branches. If a branch misprediction is detected during execute, the pipeline is flushed as usual and the misspeculated EBHR entry is cleared.

5.3 Implementing Hurdle checks

A Hcheck instruction runs after instruction decoding, as at that point we have both the pattern and the current EBHR. The check compares a branch's pattern with the current EBHR, and computes a pass-or-fail result. There is ample data parallelism within each check: the comparison of each EBHR entry with its corresponding pattern entry is independent of other entries, allowing for indirect jumps and calls to be checked quickly.

Once the check instruction reaches the retire stage, failing checks raise an exception to notify the OS. EBHR entries corresponding to mis-speculated branches thus avoid raising spurious exceptions, as any check that fails spuriously (by comparing with a mis-speculated EBHR entry) will be rolled back before it can retire.

[2]Section 4.1 describes the EBHR used by our software tracing infrastructure, which uses 3-bit EBHR entries to save space.

5.4 Sizing the EBHR

In our proposed design, the EBHR holds 8 entries (of 7 bits each) for a total size of 56 bits. There are an interesting set of trade-offs surrounding the choice of EBHR size. As the EBHR grows, the constraints we feed to the SMT solver also grow in complexity, leading to increased running time. An increase in EBHR size also requires more space to encode gate patterns in the binary and in hardware structures. For example, an EBHR that holds 64 entries would require 448 bits, making instruction encoding challenging. Increasing the EBHR can also in principle lead to improved security as longer patterns offer more opportunities to prevent chaining between gadgets. As we demonstrate in Section 7, an 8-entry EBHR is sufficient to satisfy anti-chaining constraints in our workloads, while keeping hardware overheads and SMT solving time low. Prior work similarly concluded that a history of 8 control-flow decisions was sufficient [34].

6 Experimental Setup

We evaluated Hurdle using a diverse set of workloads. We use the Autobench [32] and MiBench [35] suites which are representative of commercial embedded workloads. We also run the SPEC CPU2006 benchmark suite to show that Hurdle scales to full-size workloads. We also evaluated the training and testing process used by Hurdle on ImageMagick-7.0.6.0 using 50 images selected semi-randomly using a search engine. We emulate the EBHR using a PIN tool [33], and we record control-flow histories for an EBHR with 8 entries. For each benchmark, we record control-flow histories for each input size. We train Hurdle using the 4K input for autobench, the small inputs for MiBench, and both test and train inputs for SPEC. Once we have computed patterns from the training inputs, we then test for false positives on the largest input from each benchmark suite (4M, large, and ref respectively). For SPEC, we collect traces on the first 300 million indirect jumps and calls due to time limitations, which is equivalent to at least 28 billion instructions on all of the SPEC benchmarks measured. We note that the hardware support provided by the EBHR would also accelerate the process of gathering dynamic histories to train Hurdle. We use the Z3 SMT solver [28] to compute gate patterns based on the control-flow histories recorded on the training inputs.

We measure the runtime performance overhead of Hurdle in three steps. First, we statically link our binaries to ensure we capture all indirect jumps/calls in library code. Second, we use Dyninst [36] to statically add padding to each indirect jump/call, modeling the extra instruction cache pressure of Hcheck instructions. Due to Dyninst limitations, we can either add no instructions, or a minimum of 7 x86 instructions occupying 26B (which save/restore state and

Parameter	Configuration
Pipeline	Out-of-Order
Number of Pipline Stages	16
Branch Prediction	Hybrid Pentium-M based
Branch Target Buffer	4K entries: 1024 sets, 4-way set associative
Branch Misprediction	14 cycles
BTB Miss	7 cycles
Hcheck latency	2 cycles
Stitch latency	1000 cycles

Table 1. Haswell-like processor modeled for Runtime Overhead performance simulation

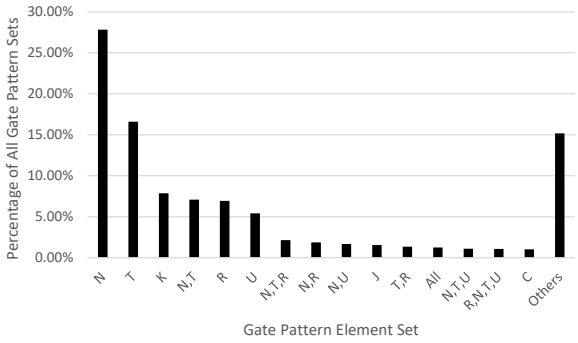

Figure 8. Gate pattern element sets used, aggregated across all applications and sorted from most used to least used. "All" is the full set of gate pattern elements, and "Others" represents the remaining 112 sets.

compute a useless result), at each indirect branch. To conservatively model each Hcheck (which carries 8B of information), we use the latter heavyweight encoding. We use the empty Dyninst instrumentation as the baseline, to account for Dyninst's static rewriting overheads (creating a copy of instrumented functions and adding trampolines from the original function). Finally, we run the rewritten binaries on SniperSim [37] (Version 6.1), modeling additional latency for Hcheck instructions on each indirect jump/call. A Haswell-based processor was used for the simulation, whose basic characteristics are outlined in Table 1. For SPEC, we limit the simulation to 1 billion instructions.

7 Evaluation

7.1 Security Guarantees

By construction, Hurdle limits the number of gadgets available for a code-reuse attack. Figure 4 shows the number of gadgets available for a code-reuse attack under Intel CET. Hurdle further reduces this number of available gadgets by ensuring that the dynamic control-flow leading up to an indirect branch must match a pattern constructed by previously seen dynamic control-flow histories. Figure 5 details the number of paths through the static control-flow graph that the Hurdle gate patterns permit. Both Hurdle and the precise dynamic control-flow histories permit less than 0.01% of all feasible static paths of length 4 through the control-flow graph. With Intel CET and Hurdle enabled, an attacker must only call EndBranch gadgets and must match the expected dynamic control-flow within a path of length 8 prior to each indirect branch or call in the attack. Further, the construction of gate patterns in Hurdle guarantees that the dynamic control-flow between gadgets will only match if required by a normal, recorded execution of the application. The combination of Intel CET [27] and Hurdle severely limits the ability of an attacker to compromise a system using a code-reuse attack.

As a brief case study, we refer to the gadgets shown in Figure 3 from `bitmnp`. These gadgets are both found in library code used by `bitmnp`. On the two bitmnp inputs, each gadget exhibits a single branch pattern. The first gadget has the pattern *NRCKUTNN*. Assuming that the EBHR matches the pattern at the call in the first gadget, the EBHR would update to *CNRCKUTN* after the call and then to *NCNRCKUT* after the direct branch in the second gadget. The second gadget has a pattern *NRKRCKNR*, which does not match contents of the EBHR after the first gadget executes. Thus, this simple attack would not be possible with Hurdle despite being possible with Intel CET.

Hurdle does not guarantee that an attacker cannot find a feasible path through Hurdle's gate patterns that permits an attack. If the attacker is able to find gadgets that match gate patterns at each indirect jump or call, Hurdle will not prevent the attack. However, Hurdle limits the ability of an attack to choose the gadgets and control-flow necessary for an effective attack.

7.2 Pattern Characterization

Figure 8 characterizes the gate patterns used by Hurdle to prevent code-reuse attacks. The graph shows the percentage that each set of elements is used across all gate patterns in all applications. Direct branches and calls account for 77.32% of all gate pattern elements. Indirect calls and jumps account for 9.29%, and returns account for the remaining 13.39% of used gate pattern elements. Overall, only 1.25% of all gate pattern elements use the full set, meaning that Hurdle can provide some constraint on program behavior the other 98.75% of the time. Of course, having no constraints on a single gate pattern element does not imply the absence of constraints on other elements in the pattern. We have no completely unconstrained patterns (composed entirely of completely unconstrained elements) among our workloads as such a pattern would by necessity permit any gadget to chain to it.

Hurdle uses one synthetic branch on each indirect jump/call. While using synthetic branches may not be strictly necessary in all cases, we found that they significantly ease the solver's task of finding satisfying patterns.

Each Hcheck instruction requires 9 bytes of additional space to encode the gate pattern and synthetic branch. For almost all applications, the additional space required for Hchecks is less than 5 KB. `xalancbmk` requires 34 KB additional space because it has 4x more indirect jumps and calls (3,848 static jumps and calls) than the next largest application (`omnetpp`). The space overhead is proportionally small compared to the initial size of the executables. Only `jpeg`, `mcf`, and `sha` have more than 1% space overhead compared to the uninstrumented binary size. `jpeg` has many indirect jump/call instructions (141), and `mcf` and `sha` have small initial binary sizes of 72.06 KB and 10.24 KB, respectively. Overall, Hurdle instrumentation adds little space overhead.

7.3 Usability Evaluation

We evaluate the usability of Hurdle by examining the process of training Hurdle to recognize the expected control-flow of an application. Hurdle must be trained on a set of inputs that are representative of the expected control-flow of a deployed application. For the `autobench`, `mibench`, and `SPEC` appplications, we evaluate a poor training environment in which Hurdle is only trained using a few inputs. We evaluate a stronger training environment using `ImageMagick`: training Hurdle to recognize the expected control-flow for two different image filters on a set of 50 images.

7.3.1 Computing Patterns

Table 2 characterizes Hurdle's use of the Z3 SMT solver [28] to compute gate patterns. For most of the evaluated applications, a satisfiable configuration of patterns can be found in less than 1 second. For the applications that take more than one second, the runtime is dominated by loading the recorded dynamic histories from file. For example, for `omnetpp`, 916.3 seconds of the runtime of the solver is dedicated to loading the required trace file. As more constraints are added, the solver tends to take more time to find a satisfying assignment.

We train on a small input and test on a larger input as described in Section 6. The larger input may cover new control flow paths that were not used by the small input and therefore are not permitted by the existing patterns. A mismatch on the existing pattern or an unknown jump/call PC will cause the Hcheck instruction to trap to software to resolve the error. Table 2 lists the number of static PCs and paths that cause a trap to software due to a mismatch or unknown PC. Hurdle uses static pattern stitching (Section 4.3.2) to determine if the error was caused by a true positive (an attack) or a false positive (lack of sufficient training data). Over all benchmarks, Hcheck instructions trap to the software stitching routine on less than .01% of all instructions. The in-memory table that Hurdle maintains for static pattern stitching consumes 8 bytes per address tag and 1 byte for the pattern. This table consumes 73.5 KB on average and a maximum of 333.8 KB on `xalancbmk`.

The software stitching routine is required for 13 of the 52 benchmarks due to either mismatched patterns or unknown jump/call PCs. We group the results for autobench into benchmarks that do not require stitching (`autobench`: bitmnp, iirflt, puwmod, tblook, ttsprk) and those that do (`autobench-fp`: aifirf, canrdr, idctrn, matrix, pntrch) because the `autobench-fp` set traps to software on shared code within the autobench suite. Four of the SPEC benchmarks (`namd`, `soplex`, `omnetpp`, and `sjeng`) trap to software stitching due to indirect jump/call PCs that are missing from the training data. `gcc`, `bwaves`, `wrf`, and `xalancbmk` trap to software stitching due to mismatches on the gate patterns computed from the training inputs.

Static pattern stitching resolves all mismatches and unknown indirect jump/call PCs in our workloads, eliminating false positives. The PC and EBHR that caused a Hcheck to fail are recorded for offline patching to avoid having to trap to software in the future. Hurdle can efficiently recompute gate patterns based on the new dynamic histories. In general, patching the gate patterns requires less time and fewer constraints than the initial computation because the existing gate patterns can be hard coded. For example, patching `omnetpp` requires only 9,764 constraints as compared to the 38,712 constraints used to compute the initial gate patterns. Thus, the SMT solver only has to compute a solution for a small number of updated branches. Once the gate patterns have been patched, no Hcheck instructions trap to software.

7.3.2 ImageMagick

`ImageMagick` was affected by multiple exploits that lead to remote code execution [38]. In these exploits, attackers leverage the Magick Vector Graphics (MVG) image format. It is likely that web services that deployed `ImageMagick` did not intend to allow the user to use the obscure MVG format, but `ImageMagick` includes the capability to use the MVG format by default. With Hurdle, a web service deploying `ImageMagick` can limit its functionality (and attack surface) in a fine-grained manner.

We evaluated Hurdle in a strong training environment using 50 image inputs on two filters: `despeckle` and `flip`. We selected these filters as examples of common operations that a web service may want to allow users to perform on images. We pick a test image, then train Hurdle on the remaining images, in alphabetical order by file name, until no Hcheck instructions trap to software. We repeat this process for each of the 50 test images, performing 50-fold cross-validation. We found that Hurdle never required the full

App	Time (s)	Constraints	Mismatch (pcs/paths)	Miss PC (pcs/paths)	App	Time (s)	Constraints	Mismatch (static/paths)	Miss PC (static/paths)
400.perlbench	66.91	6,698	0	0	465.tonto	425.59	3,888	0	0
401.bzip2	1.57	2,872	0	0	470.lbm	0.06	912	0	0
403.gcc	8.88	8,462	3 / 15	17 / 43	471.omnetpp	1,045.71	38,712	0	30 / 241
410.bwaves	0.17	1,856	1 / 1	0	473.astar	1,594.23	2,704	0	0
416.gamess	304.27	2,412	0	0	481.wrf	10.07	3,066	1 / 1	2 / 4
429.mcf	5.01	4,498	0	0	482.sphinx3	43.62	1,352	0	0
433.milc	2.55	1,110	0	0	483.xalancbmk	1,144.10	112,498	3 / 3	0
434.zeusmp	0.19	2,304	0	0	autobench	0.66	2,648	0	0
435.gromacs	18.49	1,704	0	0	autobench-fp	6.79	2,742	0	2 / 2
436.cactusADM	0.28	1,734	0	0	adpcm	0.03	450	0	0
437.leslie3d	0.24	2,034	0	0	bitcnts	4.83	722	0	0
444.namd	11.00	3,544	0	3 / 17	blowfish	0.04	528	0	0
445.gobmk	46.67	2,054	0	0	crc32	0.07	848	0	0
447.dealII	188.61	4,320	0	0	dijkstra	0.08	672	0	0
450.soplex	76.0	5,294	0	2 / 3	fft	0.13	672	0	0
453.povray	773.74	3,584	0	0	gsm	0.06	888	0	0
454.calculix	11.96	1,946	0	0	jpeg	0.26	2,720	0	0
456.hmmer	2.63	4,750	0	0	lame	0.30	2,034	0	0
458.sjeng	1,114.40	3,706	0	1 / 2	patricia	0.40	732	0	0
459.GemsFDTD	52.07	2,324	0	0	sha	0.11	1,278	0	0
462.libquantum	0.96	2,648	0	0	stringsearch	0.04	546	0	0
464.h264ref	702.24	4,793	0	0	typeset	0.98	2,208	0	0

Table 2. SMT solver statistics for initial training. Mismatch lists check failures due to mismatched gate patterns, and Miss PC lists indirect jumps/calls that are never encountered in training inputs. Hurdle uses static pattern stitching to determine whether mismatches and missing PCs are true or false positives. Hurdle records the mismatch or unknown jump/call for offline patching. After patching, all applications exhibit 0 mismatches and missing PCs.

set of 49 training images. The `despeckle` filter required a maximum of 18 training images for Hurdle to recognize expected control-flow across the 50 test images; the median number of training images was 9. The simpler `flip` filter required at most 1 training image. With a reasonable number of training inputs, Hurdle can efficiently learn the dynamic control-flow behavior of an application.

7.4 Performance Evaluation

We evaluate the performance of Hurdle's offline SMT process for computing gate patterns and the runtime performance overhead of applying Hurdle to a binary.

The Sniper Multi-Core Simulator (SniperSim) does not model indirect jumps and calls, which are a key component of the Hurdle system, so we designed our baseline experiment to model a 4K-entry Branch Target Buffer (BTB) common to all branches, direct and indirect. A 7-cycle penalty is incurred for BTB misses due to PC or Target mismatches. We normalized to the baseline architecture described in Table 1. For our experiments, we model an 8-entry EBHR to track control flow events ($\{T, N, U, K, C, J, R\}$).

Figure 9 shows the runtime overheads for applications using Hurdle. On average, Hurdle incurs only 1.02% performance overheads over the baseline execution. `stringsearch` exhibits the highest overhead because it uses 62,620 Hcheck instructions per million instructions. The baseline execution of `stringsearch` only executes for approximately 1 million cycles. This low cycle count amplifies the cost of initialization and finalization in the runtime system, which contains a high concentration of indirect calls and jumps. `fft` and `typeset` also have relatively high ratios of Hcheck instructions per million instructions. `namd` incurs a 3.4x increase in BTB misses due to the padding and instrumentation added using DynInst. We observe small speedups on `patricia`, `mcf`, and `sjeng` likely due to noise and a small improvement in instruction cache locality due to the DynInst instrumentation. If we factor out these speedups from the average, the performance overhead is still relatively low at 1.60%. We note that the results in Figure 9 for `bitcnts`, `lame`, `soplex`, and `omnetpp` do not include the performance overheads that would be incurred by additional instructions in the code due to limitations with DynInst. For these applications, we measured the performance overhead of Hcheck instructions in SniperSim on the statically linked binaries but were unable to pad the binaries to model instruction cache effects.

8 Related Work

Hurdle follows a long history of research on enforcing control-flow integrity [39]. These prior approaches can be broadly classified as heuristic-based, context-insensitive, and context-sensitive control-flow integrity.

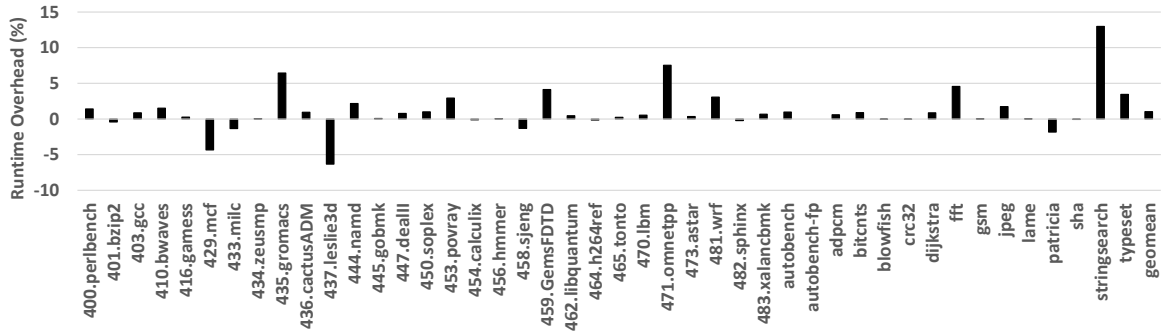

Figure 9. Runtime overheads of Hurdle as measured by SniperSim

Heuristic-based control-flow integrity systems rely on some set of heuristics to identify when the application has entered a gadget chain [22, 23]. These heuristics can be quite rich, like the parameterizable gadget chain detection based on a context-free grammar in the SCRAP system [40]. Heuristic-based systems have low performance costs and can detect known control-flow integrity exploits, but they are limited by the heuristics that are used to detect whether or not the application is executing a gadget chain. Prior work has shown that heuristic-based approaches are vulnerable to sufficiently sophisticated attacks [3–5, 41].

Traditional control-flow integrity techniques rely on static information about the control-flow of the application to determine which paths are valid [9, 11–19, 42]. For example, Control-Data Isolation [43] converts indirect branches into a series of direct branches based on the set of statically-reachable targets, using hardware support to reduce the instruction cache impact [44]. The REV system [42] records cryptographic hashes of each basic block to thwart code injection and invalid control-flow transfers, but relies on static analysis to determine basic block boundaries. These context-insensitive techniques are limited by static information about the control-flow of the application, which must be conservative. Attackers can still exploit gadgets that stay within the static CFG [4]. More recent approaches update the control-flow graph at runtime but incur some runtime overhead for these updates [20, 21].

Context-sensitive control-flow integrity uses dynamic information to improve the accuracy of detection. PathArmor, Griffin, FlowGuard, and uCFI [24–26, 45] validate execution history at sensitive system calls to detect possible gadget chains that have led to that point. We show in Section 2.5 that using the static CFG for validation permits a large number of feasible control-flow paths to each indirect branch. Hurdle further limits these feasible control-flow paths using dynamic histories, and performs checks at *every* indirect branch to thwart attacks sooner. Prior work has shown that fully-precise static CFI can still be broken [4]. uCFI incorporates dynamic information from Intel Processor Trace, but it performs checks in parallel with the application being secured, which can leave a window for the attacker to corrupt or spoof the trace data. uCFI overheads are 7% on average.

One existing approach uses the branch predictor state as part of a control-flow signature to detect attacks [34]. In this approach, the control-flow signature contains only taken and not-taken bits and combines signatures into a Bloom filter, which may introduce additional avenues for attack. This work provides no evaluation of false positives or training required for a dynamic approach to learning the control-flow behavior of an application. Further, the approach does not provide the security guarantee enforced by our SMT solver.

Other techniques have been proposed to thwart different aspects of code-reuse attacks, including ASLR [46–49] and stack guards [50]. These protection mechanisms may prevent some code-reuse attacks in practice, but they can be circumvented by sophisticated attacks [51].

Memory safety [6, 7, 7, 12, 12, 52–54] and Code-Pointer Integrity [8] guarantee safety from code-reuse attacks by preventing the memory safety vulnerability pre-condition of a code-reuse attack. Although memory safety is a stronger property that prevents both code-reuse attacks and non-control data attacks, the need to instrument memory accesses entails higher performance overheads.

9 Conclusion

This paper introduces a new hardware-based context-sensitive control-flow integrity technique, Hurdle, which builds upon Intel's forthcoming CET support for CFI. Hurdle prevents code-reuse attacks by learning an application's execution patterns and checking before each vulnerable indirect jump or call that these patterns have been respected. Hurdle uses SMT solving to infer these patterns from dynamic traces, and adds additional constraints to eliminate gadget chaining by construction. We demonstrate that Hurdle has low performance and space overheads.

Acknowledgments This work was supported by NSF grant #1525296. Opinions or findings in this material are the authors' and do not necessarily reflect the views of the NSF.

References

[1] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.

[2] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 385–399, Berkeley, CA, USA, 2014. USENIX Association.

[3] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, Washington, DC, USA, 2014. IEEE Computer Society.

[4] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 161–176, Berkeley, CA, USA, 2015. USENIX Association.

[5] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 401–416, Berkeley, CA, USA, 2014. USENIX Association.

[6] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[7] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *Proceedings of the 2018 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, pages 28:1–28:30, New York, NY, USA, June 2018. ACM.

[8] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[9] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[10] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[11] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: Hardware-assisted Flow Integrity Extension. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 74:1–74:6, New York, NY, USA, 2015. ACM.

[12] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[13] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 353–362, New York, NY, USA, 2011. ACM.

[14] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, Washington, DC, USA, 2014. IEEE Computer Society.

[15] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association.

[16] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 199–210, New York, NY, USA, 2013. ACM.

[17] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[18] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 369–382, Berkeley, CA, USA, 2013. USENIX Association.

[19] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 94–105, Washington, DC, USA, 2012. IEEE Computer Society.

[20] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 577–587, New York, NY, USA, 2014. ACM.

[21] Ben Niu and Gang Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1317–1328, New York, NY, USA, 2014. ACM.

[22] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *In Symposium on Network and Distributed System Security (NDSS*, 2014.

[23] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 447–462, Berkeley, CA, USA, 2013. USENIX Association.

[24] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, New York, NY, USA, 2015. ACM.

[25] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 585–598, New York, NY, USA, 2017. ACM.

[26] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 529–540, Feb 2017.

[27] Intel Corporation. Control-flow enforcement technology preview. Technical report, Intel Corporation, 2017.

[28] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. Technical report, Intel Corporation, 2016.

[30] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[31] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[32] EEMBC. *Autobench 2.0*, 2017. http://www.eembc.org/autobench2/index.php.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[34] Y. Shi and G. Lee. Augmenting branch predictor to secure program execution. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 10–19, June 2007.

[35] Matthew Guthaus, Jeff Ringenberg, Dan Ernst, Todd Austin, Trevor Mudge, and Richard Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[36] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[37] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[38] ImageTragick. *ImageTragick*, 2016. https://imagetragick.com.

[39] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[40] Mehmet Kayaalp, Timothy Schmitt, Junaid Nomani, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SCRAP: Architecture for signature-based protection from Code Reuse Attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 258–269, Feb 2013.

[41] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, May 2015.

[42] Erdem Aktas, Furat Afram, and Kanad Ghose. Continuous, Low Overhead, Run-Time Validation of Program Executions. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 229–241, Dec 2014.

[43] William Arthur, Ben Mehne, Reetuparna Das, and Todd Austin. Getting in control of your control flow with control-data isolation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 79–90, Washington, DC, USA, 2015. IEEE Computer Society.

[44] William Arthur, Sahil Madeka, Reetuparna Das, and Todd Austin. Locking down insecure indirection with hardware-based control-data isolation. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 115–127, New York, NY, USA, 2015. ACM.

[45] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1470–1486, New York, NY, USA, 2018. ACM.

[46] Mingwei Zhang and R. Sekar. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 91–100, New York, NY, USA, 2015. ACM.

[47] Ashish Venkat, Sriskanda Shamasunder, Dean M. Tullsen, and Hovav Shacham. HIPStR: Heterogeneous-ISA Program State Relocation. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, 2016.

[48] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[49] Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 448–459, New York, NY, USA, 2016. ACM.

[50] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[51] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[52] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM.

[53] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, New York, NY, USA, 2006. ACM.

[54] Zhengyang Liu and John Criswell. Flexible and efficient memory object metadata. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 36–46, New York, NY, USA, 2017. ACM.