Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity[*]

Isaac Evans
MIT Lincoln Laboratory
ine@mit.edu

Fan Long
MIT CSAIL
fanl@csail.mit.edu

Ulziibayar Otgonbaatar
MIT CSAIL
ulziibay@csail.mit.edu

Howard Shrobe
MIT CSAIL
hes@csail.mit.edu

Martin Rinard
MIT CSAIL
rinard@csail.mit.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

Stelios Sidiroglou-Douskos
MIT CSAIL
stelios@csail.mit.edu

ABSTRACT

Control flow integrity (CFI) has been proposed as an approach to defend against control-hijacking memory corruption attacks. CFI works by assigning tags to indirect branch targets statically and checking them at runtime. Coarse-grained enforcements of CFI that use a small number of tags to improve the performance overhead have been shown to be ineffective. As a result, a number of recent efforts have focused on fine-grained enforcement of CFI as it was originally proposed. In this work, we show that even a fine-grained form of CFI with unlimited number of tags and a shadow stack (to check calls and returns) is ineffective in protecting against malicious attacks. We show that many popular code bases such as Apache and Nginx use coding practices that create flexibility in their *intended* control flow graph (CFG) even when a strong static analyzer is used to construct the CFG. These flexibilities allow an attacker to gain control of the execution while strictly adhering to a fine-grained CFI. We then construct two proof-of-concept exploits that attack an unlimited tag CFI system with a shadow stack. We also evaluate the difficulties of generating a precise CFG using scalable static analysis for real-world applications. Finally, we perform an analysis on a number of popular applications that highlights the availability of such attacks.

[*]This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002 and Defense Advanced Research Projects Agency under Contract #FA8650-11-C-7192. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813646.

1. INTRODUCTION

Memory corruption bugs continue to be a significant problem for unmanaged languages such as C/C++ [7, 17, 53]. The level of control provided by unmanaged languages, such as explicit memory management and low level hardware control, makes them ideal for systems development. Unfortunately, this level of control, bears a heavy cost: lack of memory safety [53]. Lack of memory safety, in turn, forms the basis for attacks in the form of code injection [40] and code reuse [17, 49]. Retrofitting memory safety to C/C++ applications can introduce prohibitive overhead (up to 4x slowdown) [37] and/or may require significant programmer involvement in the form of annotations [30, 38].

As a result, the past three decades of computer security research have created a continuous arms race between the development of new attacks [12, 13, 21, 40, 49, 50] and the subsequent development of corresponding defenses [4, 19, 31, 41, 54]. This arms race attempts to strike a balance between the capabilities of the attackers and the overhead, compatibility, and robustness of the defenses [53].

The wide spread deployment of defenses such as Data Execution Prevention (DEP) [35, 41], address space layout randomization (ASLR) [54] and stack smashing protection (SSP) [19] has driven the evolution, and sophistication, of attacks. Information leakage attacks [12, 47, 52] enable the construction of multi-step attacks that bypass ASLR and SSP, while code reuse attacks, such as return-oriented program (ROP) [49], jump-oriented programming (JOP) [13], and return-to-libc [56] can be used to circumvent DEP.

The majority of the attacks rely on some form of control hijacking [53] to redirect program execution. Control Flow Integrity (CFI) is a runtime enforcement technique that provides practical protection against code injection, code reuse, and is not vulnerable to information leakage attacks [4, 61, 62]. CFI provides runtime enforcement of the intended control flow transfers by disallowing transfers that are not present in the application's Control Flow Graph (CFG). CFGs are constructed either by analyzing the source code [55], or, less accurately, by analyzing the disassembled binary [61]. The enforcement is done by assigning tags to indirect branch targets and checking that indirect control transfers point to valid tags.

Precise enforcement of CFI, however, can introduce significant overhead [4, 5]. This has motivated the development of more practical, coarse-grained, variants of CFI that have lower performance overhead but enforce weaker restrictions (i.e., limit the number of tags) [61, 62]. For example, control transfer checks are relaxed to allow transfers to *any* valid jump targets as opposed to the *correct* target. Unfortunately, these implementations have been shown to be ineffective as they allow enough valid transfers to enable an attacker to build a malicious payload [22].

As a result of the attacks on coarse-grained variants of CFI, researchers have focused on fine-grained, yet still practical enforcement of CFI. For example, forward-edge CFI [55] enforces a fine-grained CFI on forward-edge control transfers (i.e. indirect calls, but not returns). Cryptographically enforced CFI [34] enforces another form of fine-grained CFI by adding message authentication code (MAC) to control flow elements which prevents the usage of unintended control transfers in the CFG. Opaque CFI (OCFI) [36] enforces fine-grained CFI by transforming branch target checks to bounds checking (possible base and bound of allowed control transfers).

The security of fine-grained CFI techniques is contingent on the ability to construct CFGs that accurately capture the intended control transfers permitted by the application. For C/C++ applications, even with access to source code, this assumption is tenuous at best. In theory, the construction of an accurate CFG requires the use of a precise (sound and complete) pointer analysis. Unfortunately, sound and complete points-to analysis is undecidable [43]. In practice, pointer analysis can be made practical by either adopting unsound techniques or reducing precision (incomplete). Unsound techniques may report fewer connections (tags), which can result in false positives when used in CFI. Given that false positives can interfere with the core program functionality, researchers have focused on building sound but *incomplete* pointer analysis.

Incomplete analysis leads to conservative over-approximate results. The analysis will conservatively report more connections (i.e,. when two pointers *may* alias). While using incomplete pointer analysis may be sufficient for most program analysis tasks, we show that it is insufficient under adversarial scenarios. The accuracy of the pointer analysis is further exacerbated by the use of common C idioms and software engineering practices that hinder the use of accurate and scalable program analysis techniques.

We present a novel attack, *Control Jujutsu* [1], that exploits the incompleteness of pointer analysis, when combined with common software engineering practices, to enable an attacker to execute arbitrary malicious code even when fine-grained CFI is enforced. The attack uses a new "gadget" class that we call Argument Corruptible Indirect Call Site (ACICS). ACICS gadgets are pairs of Indirect Call Sites (ICS) and target functions that enable Remote Code Execution (RCE) while respecting a CFG enforced using fine-grained CFI. Specifically, ACICS gadgets 1) enable argument corruption of indirect call sites (data corruption) that in conjunction with the corruption of a forward edge pointer 2) can direct execution to a target function that when executed can exercise remote code execution (e.g., system calls). We show that for modern, well engineered applications, ACICS gadgets are readily available as part of the intended control transfer.

To demonstrate our attack, we construct two proof-of-concept exploits against two popular web servers, Apache HTTPD and Nginx. We assume that the servers are protected using fine-grained CFI (unlimited tags), to enforce only intended control transfers on the forward-edge (i.e,. indirect calls/jumps), and a shadow stack to protect the backward-edge (i.e., returns). For the forward edge, the CFG is constructed using the state-of-the-art Data Structure Analysis (DSA) [33] pointer analysis algorithm. For the backward edge, the shadow stack provides a sound and complete dynamic analysis (i.e., there is no imprecision). We show that even under this scenario, which is arguably stronger than any of the available fine-grained CFI implementations, an attacker can perform a control hijacking attack while still operating within the intended CFG.

To evaluate the prevalence, and exploitability, of ACICS gadgets, we evaluate 4 real-world applications. The results show that ACICS gadgets are prevalent and provide a rich target for attackers. Our results indicate that in the absence of data integrity, which is hard to achieve for practical applications, fine-grained CFI is insufficient protection against a motivated attacker.

This paper makes the following contributions:

- **Control Jujutsu:** We present Control Jujutsu, a new attack on fine-grained CF that exploits the incompleteness of pointer analysis, when combined with common software engineering practices, to enable an attacker to execute arbitrary malicious code.
- **ACICS gadgets:** We introduce a new "gadget" class, ACICS, that enables control hijacking attacks for applications protected using fine-grained CFI.
- **Proof-of-Concept Exploits:** We present two proof-of-concept exploits against Apache HTTPD and Nginx protected using fine-grained CFI with forward and backward-edge protection.
- **Experimental Results:** We present experimental results that characterize the prevalence of ACICS gadgets in real-world applications.

2. EXAMPLE EXPLOIT

We next present an example that illustrates how Control Jujutsu utilizes ACICS gadgets in conjunction with the imprecision of the DSA pointer analysis algorithm to create an RCE attack on Apache 2.4.12, a popular web server.

2.1 Threat Model

The threat model in this paper is a remote attacker trying to hijack control of a machine by exploiting memory vulnerabilities. We assume the system is protected by fine-grained CFI with unlimited tags for the forward edge and a shadow stack implementation for the backward edge. We also assume the deployment of DEP and ASLR. These assumptions are consistent with the literature on code reuse attacks. Finally, we assume the availability of a memory corruption vulnerability that allows an attacker to corrupt certain values on stack or heap. As numerous past vulnerabilities have shown, this assumption is realistic. It is also weaker than an arbitrary attacker read/write assumption made in the related work [31].

2.2 ICS Discovery

Control Jujutsu begins with a search for suitable ICS sites for the ACICS gadget. Control Jujutsu identifies the following requirements for ICS locations:

1. The forward edge pointer and its argument(s) should reside on the heap or a global variable to facilitate attacks from multiple data flows.

[1]Jujutsu is a Japanese martial art in which an opponent's force is manipulated against himself rather than using one's own force. In Control Jujutsu, an application's intended controls are manipulated against it.

```
1  AP_IMPLEMENT_HOOK_RUN_FIRST(apr_status_t,dirwalk_stat,
2    (apr_finfo_t *finfo,
3     request_rec *r,
4     apr_int32_t wanted),
5    (finfo, r, wanted), AP_DECLINED)
6
7  apr_status_t ap_run_dirwalk_stat(
8   apr_finfo_t *finfo, request_rec *r,
9   apr_int32_t wanted) {
10   ap_LINK_dirwalk_stat_t *pHook;
11   int n;
12   apr_status_t rv = AP_DECLINED;
13   ...
14   //check the corresponding field of the global _hooks
15   if (_hooks.link_dirwalk_stat) {
16     pHook = (ap_run_dirwalk_stat_t *)
17              _hooks.link_dirwalk_stat->elts;
18    //invoke registered functions in the array one by
19    //one until a function returns a non-decline value.
20     for(n=0; n < _hooks.link_dirwalk_stat->nelts;++n){
21       ...
22       // our seelcted ICS
23       rv = pHook[n].pFunc(finfo, r, wanted);
24       ...
25       if (rv != AP_DECLINED) break;
26     }
27   }
28   ...
29   return rv;
30 }
```

Figure 1: APR hook macro in server/request.c:97 defining `ap_run_dirwalk_stat()` in Apache HTTPD and the simplified code snippet of `ap_run_dirwalk_stat()`

```
if (r->finfo.filetype == APR_NOFILE ||
    r->finfo.filetype == APR_LNK) {
  rv = ap_run_dirwalk_stat(&r->finfo,
                           r,
                           APR_FINFO_MIN);
```

Figure 2: *dirwalk_stat* called in server/request.c:616 in Apache HTTPD

2. The arguments at the ICS can be altered without crashing the program (before reaching a target function).

3. The ICS should be reachable from external input (e.g., a network request).

Using these requirements, we found many viable ACICS candidates which we discuss at length in section 5.1. Here we present a detailed example exploit based on the selected ICS seen in Figure 1. Lines 1-5 use a macro defined in the Apache Portable Runtime (APR) library to define the function `ap_run_dirwalk_stat()`. Lines 7-30 present the simplified code snippet of `ap_run_dirwalk_stat()` after macro expansion. The actual ICS itself occurs at line 23, which invokes the function pointer `pHook[n].pFunc`. Figure 2 presents the specific `ap_run_dirwalk_stat()` call we use in our exploit.

Apache HTTPD uses a design pattern that facilitates modularity and extensibility. It enables Apache module developers to register multiple implementation function hooks to extend core Apache functionality. `ap_run_dirwalk_stat()` is a wrapper function that iteratively calls each registered implementation function for the *dirwalk* functionality until an implementation function returns a value other than `AP_DECLINED`.

2.3 Target Selection

Next, Control Jujutsu searches the application for candidate target sites for the ACICS gadgets. Control Jujutsu identifies target functions that exercise behavior equivalent to a RCE (e.g. *system* or *exec* calls).

```
/* Spawn the piped logger process pl->program. */
static apr_status_t piped_log_spawn(piped_log *pl)
{
  apr_procattr_t *procattr;
  apr_proc_t *procnew = NULL;
  apr_status_t status;

  ...
  char **args;
  apr_tokenize_to_argv(pl->program, &args, pl->p);
  procnew = apr_pcalloc(pl->p, sizeof(apr_proc_t));
  status = apr_proc_create(procnew,
                   args[0],
                   (const char * const *) args,
                   NULL, procattr, pl->p);
  ...
}
```

Figure 3: Target function *piped_log_spawn* in Apache HTTPD

In this example, the `piped_log_spawn` function meets and exceeds all of our requirements. Apache allows a configuration file to redirect the Apache logs to a pipe rather than a file; this is commonly used by system administrators to allow transparent scheduled log rotation. This functionality involves Apache reading its configuration file, launching the program listed in the configuration file along with given arguments, and then connecting the program's standard input to Apache's log output.

Figure 3 presents a simplified version of the example target function, `piped_log_spawn`. This target function accepts a pointer to the `piped_log` structure as an argument. `piped_log_spawn` invokes an external process found in the `char *program` field of the `piped_log` structure.

The `piped_log` structure has a similar layout to many other Apache structure types, which significantly expands the number of viable ACICS that can reach it without a crash. This is because many Apache structs also have an entry with type `apr_pool_t` as their first field, so that value will not need to be overwritten. This also eliminates the need to leak valid memory values for the `apr_pool_t` field, which must be valid for our example attack to succeed.

2.4 Exploit Generation:

Next, Control Jujutsu constructs the exploit as follows:

1. Use a heap memory corruption vulnerability to corrupt an entry in the `_hooks` structure's `link_dirwalk_stat` field to point to `piped_log_spawn`.

2. Use the same vulnerability to corrupt the struct in the `request_rec->finfo` field such that, when viewed as a `piped_log` struct, the fields `read_fd` and `write_fd` are null, and the field `program` points to a string with the name and arguments of the program we intend to invoke (e.g., "/bin/sh -c ...").

2.5 CFG Construction

Next, Control Jujutsu examines the CFG to ensure that the ACICS sites we identified using a tool described in Section 4 can be redirected to the target site. In our example, the CFG constructed by the DSA algorithm [33] allows the ICS located at `dirwalk_stat` to point to the target function `piped_log_spawn`. In the next section, we describe why DSA, a context-sensitive and field-sensitive analysis, was not able to construct a CFG that can be used by fine-grained CFI to stop the attack.

```
1  void ap_hook_dirwalk_stat(ap_HOOK_dirwalk_stat_t *pf,
2                                                ...) {
3    ap_LINK_dirwalk_stat_t *pHook;
4    //check the corresponding field of the global _hooks
5    if (!_hooks.link_dirwalk_stat)
6      _hooks.link_dirwalk_stat = apr_array_make(...);
7    // store the function pointer pf into the array
8    pHook = apr_array_push(_hooks.link_dirwalk_stat);
9    pHook->pFunc = pf;
10   ...
11 }
```

Figure 4: The code snippet for `ap_hook_dirwalk_stat()` in Apache HTTPD

3. BUILDING CONTROL FLOW GRAPHS WITH STATIC ANALYSIS

The construction of a precise CFG requires a pointer analysis [6, 23, 24, 26, 33, 42, 45, 51, 59] to determine the set of functions to which the pointer at each indirect call site (e.g., line 23 in Figure 1) can point.

Figure 4 presents a simplified version of `ap_hook_dirwalk_stat()`, which registers implementation functions that `ap_run_dirwalk_stat()` (shown in Figure 1) can later invoke for the functionality of *dirwalk_stat*. The intended behavior of the ICS shown at line 23 in Figure 1 is to only call implementation functions registered via `ap_hook_dirwalk_stat()` in Figure 4.

The first argument `pf` of `ap_hook_dirwalk_stat()` is the function pointer to an implementation function of *dirwalk_stat*. It has the type `ap_HOOK_dirwalk_t`, which corresponds to the function signature for *dirwalk_stat*. `ap_hook_dirwalk_stat()` stores the function pointer to the APR array `_hooks.link_dirwalk_stat`. `ap_LINK_dirwalk_stat_t` (line 3 in Figure 1) represents the type of each array entry.

The function `ap_run_dirwalk_stat()` (line 7 in Figure 1) iterates over the APR array `_hook.link_dirwalk_stat` and runs each implementation function until an implementation function returns a value other than `AP_DECLINED`.

The example code in Figure 1 and Figure 4 highlights the following challenges for the static analysis:

- **Global Struct:** The analysis has to distinguish between different fields in global variables. `_hooks` in Figure 1 and Figure 4 is a global struct variable in Apache HTTPD. Each field of `_hooks` contains an array of function pointers to registered implementation functions for a corresponding functionality. For example, the `link_dirwalk_stat` field contains function pointers to implementation functions of the functionality *dirwalk_stat*.

- **Customized Container API:** The analysis has to capture inter-procedural data flows via customized container APIs. The code in Figure 1 and Figure 4 uses customized array APIs `apr_array_push()` and `apr_array_make()` to store and manipulate function pointers.

- **Macro Generated Code:** The code shown in Figure 1 and Figure 4 is generated from macro templates found in Apache Portable Runtime library. For example, for a functionality *malicious*, there are pairs of functions `ap_hook_malicious()` and `ap_run_malicious()` that are structurally similar to the code shown in Figure 1 and Figure 4. This imposes a significant additional precision requirement on the static analysis, as it needs to consider a (potentially) large number of similar functions that can manipulate the data structures inside `_hooks`.

3.1 Static Analysis: Knobs and Trade-offs

Precise (sound and complete) pointer analysis is undecidable [43]. Unsound pointer analysis may generate a CFG that misses legitimate indirect transfers, which may ultimately lead CFI to report false positives. Breaking program functionality is typically undesirable (see Section 3.3).

Researchers instead focus on sound but incomplete pointer analysis algorithms [6, 23, 24, 26, 33, 42, 45, 51, 59] that conservatively report more connections. For example, two pointers *may* alias and an indirect call site *may* call a function. The hope is that such imprecision could be controlled and that the analysis could be accurate enough so that the generated CFG still does not contain malicious connections.

Another important design decision for pointer analysis algorithms is scalability [25]. Standard pointer analysis algorithms for C programs have three important knobs that control the trade-offs between accuracy and scalability: context-sensitivity, field sensitivity, and flow sensitivity.

Context Sensitivity: A context-sensitive analysis [33, 51, 59] is able to distinguish between different invocations of a function at different call sites. It tracks local variables, arguments, and return values of different function invocations, at different call sites separately. A context-insensitive analysis, in contrast, does not distinguish between different invocations of a function, i.e., analysis results for local variables, arguments, and the return values from different invocations of the function are merged together.

Previous work in the programming language community has shown that context sensitivity is indispensable for obtaining precise pointer analysis results in real world applications [25, 33, 51, 59], because it eliminates a large set of unrealizable information propagation paths where calls and returns do not match. Context sensitivity is especially important for analyzing C programs that implement customized memory management functions or manipulate generic data structures with common interfaces, because otherwise all pointer values returned by each memory management or data structure function will be aliased (to each other).

For our example in Figure 4, context sensitivity is also important. A context-insensitive analysis will merge the analysis results of the return value of different invocations of `apr_array_push()`. Therefore a context-insensitive analysis will incorrectly determine that `pHook` at line 9 in Figure 4 may alias to `pHook` in another implementation registration function such as `ap_hook_malicious()` (recall that all implementation registration functions are generated with macro templates), because both are equal to a return value of `apr_array_push()` (albeit from different invocations). Eventually, this imprecision will cause the analysis to determine that the indirect call at line 23 in Figure 1 may call to an implementation function registered via `ap_hook_malicious()`, because the analysis conservatively determines that the function pointer argument value in `ap_hook_malicious()` may flow to `pHook->pFunc` via the aliased `pHook` pointer.

Unfortunately, context-sensitive pointer analysis is expensive for large real-world applications. Full context-sensitive analysis is also undecidable for programs that contain recursions [44]. Standard clone-based context-sensitive pointer analysis [59] duplicates each function in a program multiple times to distinguish different invocations of the function. This unfortunately will increase the size of the analyzed program exponentially. The DSA algorithm uses bottom-up and top-down algorithms to traverse the call graph of a program and summarizes context-sensitive analysis results into a unification-based data structure graph [33]. It produces slightly less accurate results than clone-based algorithms but avoids an exponential blow up on real world programs.

Field Sensitivity: A field-sensitive analysis [33, 42] is able to distinguish different fields of a struct in C programs, while a field-insensitive analysis treats the whole struct as a single abstract variable. Modifications to different fields are transformed into weak updates to the same abstract variable, where the analysis conservatively assumes that each of the modifications *may* change the value of the abstract variable.

For our example in Figure 1 and Figure 4, field sensitivity is important. A field insensitive analysis treats the global struct `_hooks` as a single abstract variable, so that it cannot distinguish the field `link_dirwalk_stat` from other fields in `_hooks` such as `link_malicious`. Therefore the analysis conservatively determines that the assignment at lines 16-17 in Figure 1 may retrieve an array that contains function pointers for other functionalities like *malicious*. This causes the analysis to eventually determine that the indirect call at line 23 in Figure 1 may make call to any implementation function registered via `ap_hook_malicious()`.

Field-sensitive pointer analysis is hard for C programs due to the lack of type-safety. Pointer casts are ubiquitous, and unavoidable for low-level operations such as `memcpy()`. Field-sensitive analysis algorithms [33, 42] typically have a set of hand-coded rules to handle common code patterns of pointer casts. When such rules fail for a cast of a struct pointer, the analysis has to conservatively merge all fields associated with the struct pointer into a single abstract variable and downgrade into a field-insensitive analysis for the particular struct pointer.

Flow Sensitivity: A flow-sensitive analysis considers the execution order of the statements in a function [23, 26, 45], while a flow-insensitive analysis conservatively assumes that the statements inside a function may execute in arbitrary order. Flow sensitivity typically improves pointer-analysis accuracy but when combined with context-sensitive analysis it can lead to scalability issues. To the best of our effort, we are unable to find any publicly available context-sensitive flow-sensitive pointer analysis that can scale to server applications such as Apache HTTPD. A common practice to improve the accuracy of a flow-insensitive analysis is to apply single static assignment (SSA) transformation to a code before the analysis [24].

3.2 DSA Algorithm

As discussed above, the combination of context sensitivity and field sensitivity is critical for generating a precise CFG that can stop the attack described in Section 2. We next present the results of using the DSA algorithm [33] to generate a CFG for Apache HTTPD. We chose the DSA algorithm because, to the best of our knowledge, it is the only analysis that 1) is context-sensitive and field-sensitive, 2) can scale to server applications like Apache HTTPD and Nginx, and 3) is publicly available.

The DSA algorithm is available as a submodule of the LLVM project [3] and is well maintained by the LLVM developers. It works with programs in LLVM intermediate representation (IR) generated by the LLVM Clang compiler [2]. We use Clang to compile Apache HTTPD together with the Apache Portable Runtime(APR) library [1] into a single bitcode file that contains LLVM IRs for the whole Apache HTTPD and APR library. We run the LLVM *mem2reg* pass (SSA transformation pass) on the bitcode file to improve the accuracy of the pointer analysis . We then construct an LLVM pass that runs the DSA algorithm and queries the DSA result to generate a CFG for the bitcode file.

Unfortunately, the DSA algorithm produces a CFG that cannot stop the attack in Section 2. Specifically, the CFG specifies that the indirect call at line 26 in Figure 4 may call to the function `piped_log_spawn()`. We inspected the debug log and the intermediate pointer analysis results of the DSA algorithm. We found that although as a context-sensitive and flow-sensitive analysis the DSA algorithm should theoretically be able to produce a precise CFG to stop the attack, the algorithm in practice loses context sensitivity and flow sensitivity because of convoluted C idioms and design patterns in Apache HTTPD and the APR library. As a result, it produces an imprecise CFG. Fine-grained CFI systems that disallow the calling of functions whose address is not taken can prevent the proposed attack through `piped_log_spawn()`. The attack can succeed, however, by targeting `piped_log_spawn()` indirectly through functions such as `ap_open_piped_log_ex()`, whose address is directly taken by the application. Next, we describe some of the sources of imprecision in more detail.

Struct Pointer Casts: We found that struct pointer-cast operations in Apache HTTPD cause the DSA algorithm to lose field sensitivity on pointer operations. Pointer casts are heavily used at the interface boundaries of Apache components. There are in total 1027 struct pointer conversion instructions in the generated bitcode file of Apache HTTPD.

For example, pointers are cast from `void*` to `apr_LINK_dirwalk_stat_t *` at line 8 in Figure 4 when using the array container API `apr_array_push()`. Apache HTTPD also uses its own set of pool memory management APIs and similar pointer casts happen when a heap object crosses the memory management APIs. When the DSA algorithm detects that a memory object is not accessed in a way that matches the assumed field layout of the object, the algorithm conservatively merges all fields into a single abstract variable and loses field sensitivity on the object.

Integer to Pointer Conversion: Our analysis indicates that the Clang compiler generates an integer to pointer conversion instruction (`inttoptr`) in the bitcode file for the APR library function `apr_atomic_casptr()`, which implements an atomic pointer compare-and-swap operation.

For such `inttoptr` instructions, the DSA algorithm has to conservatively assume that the resulting pointer may alias to any pointers and heap objects that are accessible at the enclosing context. Although such instructions are rare (`apr_atomic_casptr()` is called three times in the Apache HTTPD source code), they act as sink hubs that spread imprecision due to this over-conservative aliasing assumption.

Cascading Imprecision: The struct pointer casts and integer to pointer conversions are the root sources of the imprecision. One consequence of the imprecision is that the DSA algorithm may generate artificial forward edges (calls) for indirect call sites.

Although initially such artificial forward edges may not directly correspond to attack gadgets in the Apache HTTPD, they introduce artificial recursions to the call graph. Because maintaining context sensitivity for recursions is undecidable, the DSA algorithm has to conservatively give up context sensitivity for the function calls between functions inside a recursive cycle (even they are artificially recursive due to the analysis imprecision). This loss of context sensitivity further introduces imprecision in field sensitivity because of type mismatch via unrealizable information propagation paths.

In our Apache HTTPD example, this cascading effect continues until the DSA algorithm reaches an (imprecise) fix-point on the analysis results. As a result, 51.3% of the abstract struct objects the DSA algorithm tracks are merged into single abstract variables (i.e., the loss of field sensitivity); we observed a phenomenal artificial recursion cycle that contains 110 functions (i.e., due to the loss of context sensitivity). Some of this imprecision may be attributed to changes in LLVM IR metadata since version 1.9. Previous versions relied on type annotations that used to persist from the llvm-gcc front-end into the LLVM IR metadata that are no longer available. LLVM DSA prior to version 1.9 used a set of type-based heuristics to improve the accuracy of the analysis. Aggressive use of type-based heuristics is unsound and could introduce false negatives (opening up another possible set of attacks).

3.3 Unsound Analysis with Annotations

To maintain soundness guarantees, existing pointer analysis algorithms conservatively over-approximate results. For example, sound pointer analysis algorithms conservatively assume that two pointers *may* alias or an indirect call site *may* call a function when analyzing hard-to-analyze C idioms or code design patterns.

One way to improve the security of fine-grained CFI is to generate CFGs using pointer analysis algorithms that relax soundness guarantees. Unsound pointer analysis can avoid such over-conservative assumptions and generate restrictive CFGs that may stop attacks based on ACICS gadgets. One consequence of applying unsound analysis, however, is that a restrictive CFG may cause undesirable false positives that interfere with legitimate program operation.

Our experiments show that developers adopt design patterns that improve modularity and maintainability at the cost of adding program analysis complexity. One way to improve pointer-analysis precision, is to rely on programmers to provide annotations that help the underlying analysis navigate hard-to-analyze code segments. One promising research direction is the design of an annotation system that improves the underlying pointer analysis with minimal developer involvement.

4. ACICS DISCOVERY TOOL

We next discuss how to automate the discovery of ACICS gadgets using the ACICS Discovery Tool (ADT). To help discover candidate ICS/target function pairs (ACICS gadgets), ADT dynamically instruments applications using the GDB 7.0+ reverse debugging framework. For each candidate ACICS gadget, ADT runs a backward data-flow analysis that discovers the location of the ICS function pointer (and its arguments) in memory. Once a candidate pair is identified, ADT automatically corrupts the forward edge pointer and its arguments to verify that remote code execution can be achieved. Below, we describe ADT's approach in detail.

4.1 Approach

As input, ADT takes a target program, a list of candidate indirect call sites (ICS), sample inputs that exercise the desired program functionality (and the list of ICS), and the address of a candidate target function inside the target program. For each ICS location, ADT performs the following steps (illustrated in Figure 5):

1. **Reach ICS:** ADT instruments program execution, using the GDB framework, with the ability to perform reverse execution analysis once program execution reaches a candidate ICS location. Specifically, ADT adds a breakpoint which enables the process recording functionality at the entry to the function enclosing the ICS location.

2. **Backward Dataflow Analysis:** Once execution reaches the ICS location, ADT performs a backward reaching-definition dataflow analysis (see Section 4.2) from the registers containing the target function address and its arguments to the memory locations that hold their values.

3. **Determine Last Write IP:** Next, ADT needs to identify program locations that can be used to corrupt the ICS function pointer and its values. To do this, ADT restarts the debugger and instruments the memory addresses, identified in the previous step, to record the code locations (i.e., the instruction pointer) that perform memory writes to these locations. To differentiate memory writes that occur in loops, ADT maintains a write counter. Using this information, ADT can determine the ideal program location to corrupt the ICS target and its arguments such as to minimize possible interference.

4. **Corrupt Function Pointers and Arguments:** At this point, ADT is able to restart the debugger and halt the program at the ideal point identified in the previous step. Then ADT redirects the ICS function pointer and its arguments to the target function. Additionally, by tracking every statement executed until the target ICS is reached, a lower bound of the liveness of the ACICS can be reported.

   The liveness of an ACICS allows us to reason about its exploitability; if the liveness persists across the program lifecycle, the ICS can be attacked by almost any memory read/write vulnerability, regardless of where it occurs temporarily. On the other hand, an ACICS whose liveness is contained in a single function is significantly less exploitable.

5. **ACICS validation:** Finally, ADT validates the ACICS gadget by verifying that the target function is reached, the argument values match the values in the corruption step and ultimately verifying that the target function can exercise functionality equivalent to remote code execution (e.g., create a file, launch a process, etc.).

```
Input  : The target ICS instruction $icsinst$.
Input  : $Prev$, a function that returns the previous instruction (or NULL if not
         available) before a given instruction.
Output: The memory address that stores the call target or NULL if failed
if $icsinst$ is the form of "call REG[$i$]" then
    $r \longleftarrow i$
else
    return NULL
$inst \longleftarrow Prev(icsinst)$
while $inst \neq NULL$ do
    if $inst$ modifies REG[$r$] then
        if $inst$ is the form of "REG[$r$] $= a *$ REG[$i$] $+ c$" then
            $r \longleftarrow i$
        if $inst$ is the form of "REG[$r$] $= *(a *$ REG[$i$] $+ c)$" then
            return $a \times regv(i) + b$
    $inst \longleftarrow Prev(inst)$
return NULL
```

Figure 6: Backward dataflow analysis to identify the target address

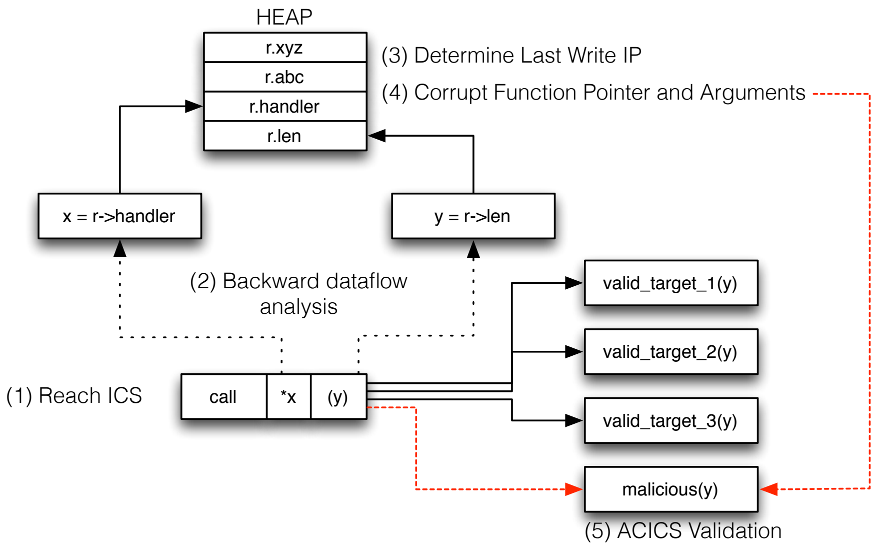

Figure 5: ACICS Discovery Tool

4.2 Backward Dataflow

Figure 6 presents ADT's backward dataflow analysis algorithm. The goal of this analysis to perform a backward reaching definition analysis from the register values that hold the target function and its arguments to corruptible memory locations. For example, in Figure 5, the dataflow algorithm called on input `x` would produce the address of `r.handler`. This is done by iteratively stepping back in time (reverse debugging) and examining any instruction that modifies the register which originally contained the function pointer. We assume that the instructions involved in the dataflow of the target function can be represented as the composition of linear functions and dereferences, and report a dataflow error if this does not hold. Once a function which dereferences a memory location is discovered, linear function models are used to compute the source address of the forward edge.

ADT contains several additional checks, such as an assertion to ensure that the forward edge pointer value at the ICS matches the value observed at the computed source memory address which is the output of the backward dataflow procedure. The typical use case discovered by the backward analysis is the lookup of a member element from a struct pointer; such as `x->y`; additional levels of indirection such as `x->y->z` are currently not supported.

4.3 Discussion

ADT was not designed to discover all possible ACICS gadgets but rather as a tool to facilitate the construction of proof-of-concept exploits. Specifically, ADT under-reports the number of ACICS gadgets for the following reasons. First, the backward dataflow analysis does not support multi-level argument redirection. Second, ADT assumes deterministic execution; non-deterministic behavior will result in under-reporting ACICS gadgets (i.e., it may miss ACICS gadgets but it will not report incorrect results). Third, ADT does not consider possible ACICS gadgets caused by unintentional arguments–pointers left in registers from previous function calls which might become relevant again if a function pointer were changed to point at a higher function of higher arity. Finally, we acknowledge that increasing the distance from ACICS gadget to target system calls may introduce more opportunities for failure due to argument clobbering. Our results show that in practice this is not a problem. Software engineering techniques such as refactoring make this less of a problem in large, well engineered software.

5. EVALUATION

We evaluate Control Jujutsu using two proof-of-concept exploits against two popular web servers Apache and Nginx. We assume that the servers are protected using fine-grained CFI (unlimited tags), to enforce only intended control transfers on the forward-edge (i.e,. indirect calls/jumps), and a shadow stack to protect the backward-edge (i.e. returns). For the forward edge, the CFG is constructed using the state-of-the-art DSA [33] pointer analysis algorithm. To protect the backward edge, we assume a shadow stack implementation.

For each exploit, we evaluate the availability of ACICS gadgets by measuring 1) the number of suitable indirect call sites and 2) the number of target functions that can be used together to launch remote code execution attacks.

5.1 Apache HTTPD 2.4.12

5.1.1 Suitable ICS:

Our evaluation of the unoptimized Apache binary shows that the server contains 172 indirect call sites (ICS). We limit our evaluation to the core binary and omit reporting potential ICS target in other Apache modules, such as the Apache Portable Runtime (APR) and APR-util libraries. From these 172 sites, we want to find a subset of sites 1) which are exercised when the program processes a request and 2) whose forward edge pointer and arguments can be successfully corrupted by our ADT tool without crashing the program.

We run our ADT tool described in Section 4 on each of the 172 sites. We use a test script program that sends simple HTTP GET requests to drive our experiments. There are 51 sites exercised in our experiments. The remaining 121 sites do not satisfy our requirement, because they are either inside specific modules that are not enabled by default or depend on specific functionalities that a simple HTTP GET request does not exercise.

Table 1 presents the classification results of ICS exercised during different execution stages of Apache. In order to detect whether an ICS is exercised during the HTTP GET request life cycle or the startup, we vary when the test script is called in our tool. Our results

Total ICS	172
Exercised in HTTP GET request	20
Exercised during startup	45
Unexercised	121

Table 1: Indirect Call Sites Dynamic Analysis

Number of ICS dynamically encountered	51
Detected forward edge pointer on the heap/global	34
Automatically corrupted forward edges	34
Automatically corrupted forward edges + arguments	3

Table 2: Automatic Corruption Analysis

show that there are 20 sites exercised during an HTTP GET request life cycle and 45 sites exercised during startup. Note that some of sites exercised during startup are also exercised by an HTTP GET request .

We use our ADT tool to detect the location of the forward edge pointer and arguments of each of the exercised 51 ICS and to corrupt these values. Table 2 presents our experimental results. Of the 51 ICS that are exercised dynamically in our experiments, our tool successfully corrupt forward edge pointers for 34 ICS. For 3 ICS our tool successfully corrupted both the forward edge pointers and the arguments.

Code patterns inside Apache facilitate our attack. We discovered that 108 of the total 172 ICS listed are in the Apache binary, but generated from the APR library's "hook" system, which allows a function to register a function pointer for a callback at a later time. For all of the ICS generated by the APR hooks, the forward edge pointers are stored inside the global struct `_hooks` inside APR (see Section 2, Figures 1 and 4). This hook structure persists across the lifetime of the Apache worker process, which is ideal for our attack. Additionally, almost all of the hook functions have argument that are pointers to objects visible across the entire request lifecycle, such as the ubiquitous `request_rec* r` argument. This is also ideal for corruption purposes.

In our Apache exploit example in Section 2, we use the ICS inside `ap_run_dirwalk_stat()`, the function meets all of our requirements and it is exercised during every HTTP GET request. While our evaluation focuses on unoptimized binaries to facilitate the construction of our proof-of-concept attacks, we also verified that the target ACICS gadget is still present in LLVM -02 level of optimizations. We believe that optimizations such as inlining will not significantly reduce the number of available gadgets.

5.1.2 Target Functions:

We run a script that searches the Apache source code for system calls that we can use to trigger behaviors equivalent to RCE such as `exec()` and `system()`. For each function in Apache, the script measures the distance between the function and a function that contains such system calls.

Table 3 presents the results. The farther away a target function is in the CallGraph, the harder it generally is to use it in the payload. At the same time, more viable functions become available. Related work has found similar results for the Windows platform [22]. Our example Apache exploit in Section 2 uses `piped_log_spawn()`, which is two calls away from the system call.

Direct calls to system calls	1 call away	2 calls away
4	13	31

Table 3: Target Functions Count Based on CallGraph distance

5.2 Nginx 1.7.11

Our analysis for Nginx mirrors the analysis we performed for Apache source code. We used the ACICS Discovery Tool (ADT) described in Section 4 and performed manual analysis to find the most suitable indirect call site and target function to demonstrate our attack.

5.2.1 Suitable ICS:

Our analysis on the unoptimized Nginx binary shows that there are 314 ICS in Nginx. We run our ADT tool on each of the 314 ICS in a way similar to our Apache experiments. Table 4 presents the classification results of ICS based on different execution stages and Table 5 presents the corruption experiment results.

Our results show that there are 36 ICS exercised during our Nginx experiments and 27 of these ICS are exercised during an HTTP GET request lifecycle after Nginx startup. Of the 36 exercised ICS, our ADT tool successfully corrupted the forward edge pointers and arguments for 4 ICS.

Total ICS	314
Exercised in HTTP GET request	27
Exercised during startup	18
Unexercised	278

Table 4: Indirect Call Sites Dynamic Analysis

Number of ICS dyanmically encountered	36
Detected forward edge pointer on the heap/global	7
Automatically corrupted forward edges	7
Automatically corrupted forward edges + arguments	4

Table 5: Automatic Corruption Analysis

We found that the ICS at core/ngx_output_chain.c:74 in `ngx_output_chain()` is an ideal candidate ICS for our attack. Figure 7 presents the simplified code snippet of `ngx_output_chain()`. The ICS is at line 27 in Figure 7. The function implements the filter chaining mechanism that is inherent to Nginx's modular design because it gives an easy way to manipulate the output of various handlers run on the request object to generate a response.

In this function, the function pointer `ctx->output_filter` and arguments `ctx->filter_ctx` are all derived from `ctx` which is a `ngx_output_chain_ctx` struct pointer. This `ctx` a global object lives on the heap, so that our tool successfully corrupts all of these values.

Secondly, the argument `ctx->filter_ctx` is a void pointer that is written only once during the request life cycle, whereas argument `in` is a pointer to the head of a linked list of filters that are applied to request responses. This linked list is modified in every module that implements a filter. However with manual dataflow analysis, it is possible to modify this linked list so that the checks at lines 18, 19, and 20 of Figure 7 pass and we reach the execution of the ICS before any crash happens. Thirdly, as all response body filters are called before the response is returned to the user, we were able to remotely exercise this ICS during the request life cycle.

```
1  ngx_int_t
2  ngx_output_chain(ngx_output_chain_ctx_t *ctx,
3                   ngx_chain_t *in)
4  {
5    ...
6
7    if (ctx->in == NULL && ctx->busy == NULL)
8    {
9      /*
10      * the short path for the case when the ctx->in
11      * and ctx->busy chains are empty, the incoming
12      * chain is empty too or has the single buf
13      * that does not require the copy
14      */
15
16     if (in == NULL) {
17       return ctx->output_filter(ctx->filter_ctx, in);
18     }
19
20     if (in->next == NULL
21 #if (NGX_SENDFILE_LIMIT)
22         && !(in->buf->in_file && in->buf->file_last
23           > NGX_SENDFILE_LIMIT)
24 #endif
25           && ngx_output_chain_as_is(ctx, in->buf))
26     {
27       return ctx->output_filter(ctx->filter_ctx, in);
28     }
29   }
30   ...
31 }
```

Figure 7: ACICS for Nginx found in *ngx_output_chain* function

5.2.2 *Target Function:*

We use a script to search Nginx source code for system calls with RCE capability. Table 6 shows the number of potential targets based on the distance in the call graph. We found that the function `ngx_execute_proc()` (shown in Figure 8) is an ideal target function for our proof-of-concept attack, because it executes a `execve()` call with passed-in arguments and it has a small arity of 2, which facilitates the type punning.

```
1  static void
2  ngx_execute_proc(ngx_cycle_t *cycle, void *data)
3  {
4    ngx_exec_ctx_t  *ctx = data;
5
6    if (execve(ctx->path, ctx->argv, ctx->envp) == -1) {
7      ngx_log_error(...);
8    }
9    exit(1);
10 }
```

Figure 8: Nginx Target Function that calls *execve*

5.2.3 *Proof-of-concept Attack:*

Hence, we identified the ACICS gadget pair for our attack which is composed of the ICS at core/ngx_output_chain.c:74 in `ngx_output_chain()` (see line 27 in Figure 7) and the target function `ngx_execute_proc()` (see Figure 8).

We then perform the attack as follows. We corrupt `ctx->output_filter` to point to the target function `ngx_execute_proc()` and we corrupt the memory region that `in` points to so that when the memory region is viewed as a `ngx_exec_ctx_t` struct in `ngx_execute_proc()`, it will trigger RCE at line 6 in Figure 8. We successfully achieved RCE with our attack.

Direct calls to system calls	1 call away	2 calls away
1	2	3

Table 6: Target Functions Count Based on CallGraph distance

5.3 CFG Construction Using DSA

We next evaluate the precision of CFG construction using the DSA algorithm on four popular server applications: Apache HTTPD, Nginx, vsftpd, and BIND. Specifically, we evaluate the loss of context sensitivity by measuring the maximum size of strongly connected components and the loss of field sensitivity by measuring the number of merged objects. We performed all of our experiments on an Intel 2.3GHz machine running Ubuntu 14.04.

Table 7 summarizes the results. The first column presents the application name. The second and third columns represent the source code line count and LLVM IR count respectively. The application size ranges from 17K LoC for vsftpd to approximately 460K LoC for BIND.

The fourth column presents the number of functions in the largest (potentially artificial) recursion cycle DSA algorithm found for each application. High numbers translate to high loss of context sensitivity. The fifth column presents the percentage of the abstract struct objects that the DSA algorithm tracks which the DSA algorithm merges conservatively. High percentage numbers indicate high loss of field sensitivity.

Together, columns four and five show that the DSA algorithm is unable to produce satisfactory results on any of the four applications due to the loss of field sensitivity and context sensitivity. DSA loses field sensitivity on up to 70.5% of tracked struct objects and detects artificial recursion groups that contain up to 1023 functions.

Note that even for Nginx, where the relative loss is small, the generated CFG is unable to stop the ASICS gadgets index Section 5.2. The CFG allows the ICS found in core/ngx_output_chain.c:74 (line 27 in Figure 7) to call the target function `ngx_execute_proc` shown in Figure 8 due to the pointer analysis imprecision.

The sixth column presents the running time of the DSA algorithm on each application. Our results show that the running time of the DSA algorithm grows non-linearly to the amount of analyzed code. For BIND, the algorithm needs more than 14 minutes to finish. This result highlights the difficult trade-offs between the accuracy and the scalability in pointer analysis algorithms.

5.4 Summary

In summary, our results demonstrate that the availability of ACICS gadgets inside Apache and Nginx can be harnessed to produce with two proof-of-concept attacks. Our results also show that on all evaluated applications, the DSA algorithm loses a significant part of field sensitivity and context sensitivity and that the generated CFGs are not precise enough to stop the proof-of-concept attacks. Together the results indicate the difficulty of creating a sound, precise and scalable CFG construction algorithm that can be used by fine-grained CFI to stop ACICS gadgets.

6. DISCUSSION

In this section, we discuss possible defenses against the Control Jujutsu attack and explore their viability, security, and practicality.

Program	LoC	LLVM IR	Max. SCC Size	Merged%	Time
HTTPD	272K	318K	110	51.3%	14s
Nginx	123K	358K	38	10.8%	10s
vsftpd	16K	24K	255	70.5%	1s
BIND	462K	1167K	1023	41.2%	14m52s

Table 7: DSA analysis statistics

6.1 Complete Memory Safety

Complete memory safety techniques that enforce both temporal and spatial safety properties can defend against all control hijacking attacks, including Control Jujutsu. Softbound with its CETS extensions [37] enforces complete memory safety albeit at a significant cost (up to 4x slowdown).

On the other hand, experience has shown that low overhead techniques that trade security guarantees for performance (e.g., approximate [48] or partial [5] memory safety) are eventually bypassed [16, 22, 47]. CPI [31] is a recent technique that achieves low performance overhead by providing memory safety properties for code pointers only (i.e., not data pointers). Unfortunately, it has already been shown to be bypassable [21].

Hardware support can make complete memory safety practical. Intel memory protection extensions (MPX) [29] can provide fast enforcement of memory safety checks. The Low-Fat fat pointers scheme shows that hardware-based approaches can enforce spatial memory safety at very low overhead [32]. Tagged architectures and capability-based systems such as CHERI [58] can also provide a promising direction for mitigating such attacks.

6.2 Runtime Arity Checking

The recently published Indirect Function-Call Checks (IFCC) [55] is forward-edge enforcement variant of CFI designed for C++ programs. In addition to forward-edge enforcement, it further imposes a restriction that the arity of call sites and target functions must match. IFCC is capable of more powerful restrictions, but they limit themselves to checking arity for reasons discussed in Section 6.3.1.

IFCC may limit the number of available ACICS, but it cannot prevent the Control Jujutsu attack in general. In particular, using our ACICS discovery tool, we were able to easily expand on our original exploit for Apache and develop an additional full exploit based on an ACICS with an arity that matches its ICS with its target function. This exploit would not be detected by IFCC and is detailed in Section 6.3.1. As for Nginx, our proof-of-concept exploit uses an ACICS gadget with matching arity so IFCC will not be able to detect it.

6.3 Runtime Type Checking (RTC)

One way to restrict ACICS gadgets is to use a runtime type checker for C. The most precise runtime type checker would need access to the program source for type name information that is typically removed by C compilers. Although some information (e.g., the width in words of arguments) is inferrable purely from binary analysis with the use of an interpreter and runtime environment, as in the Hobbes checker [14], but the guarantees of runtime type checking are substantially weakened.

6.3.1 Challenges of RTC

Unfortunately, runtime checks based on source code inference would break compatibility with a large subset of real-world code. Qualifiers such as *const* are routinely violated at runtime; a recent paper [18] found that for *const* pointers alone, each of thirteen large FreeBSD programs and libraries examined contained multiple "de-const" pointer idioms which would be broken if *const* had been enforced at runtime. In general, real-world programs do not always respect function pointer types at runtime, as the IFCC paper noted when they explained that their approach could support one tag per type signature, but that this "can fail for function-pointer casts."

The callback and object-oriented programming patterns that exist in large C programs are analogous to the virtual table semantics of C++ programs. As our attack examples clearly demonstrate, these indirect call sites in C programs with higher-order constructs require protections in the same way that C++ programs need principled virtual table protection.

A telling example of these patterns is the APR library's bucket brigade system. The bucket brigade structure, seen in Figure 9 is analogous to a C++ object. It contains members like "data" along with generic member functions that know how to read, write, and delete the data. Additionally, buckets live on the heap, so they are globally visible and thus can be corrupted in any function with a heap vulnerability.

```
struct apr_bucket_type_t {
  const char *name;
  int num_func;
  void (*destroy)(void *data);
  ...
};

struct apr_bucket {
   const apr_bucket_type_t *type;
   apr_size_t length;
   apr_off_t start;
   void *data;
   void (*free)(void *e);
   ...
};
```

Figure 9: *bucket_brigade* declarations in APR-util

The structure is exercised by macros in the APR-util library such as the `bucket_brigade_destroy` seen in Figure 10. This macro is an ideal example of an ACICS–particularly dangerous because the function it executes and its argument are stored in a closure-like style inside the same structure. If an attacker can corrupt a bucket brigade struct which is later destroyed, an arbitrary function can be called with an arbitrary argument.

There are dozens of calls to `apr_bucket_destroy` and its wrapper macro `apr_bucket_delete` in the Apache source. We verified that the DSA analysis determines that `apr_bucket_delete` might call `piped_log_spawn`. Unlike the example in Figure 2, the arities of the ICS and the target match, which passes the arity check imposed by IFCC.

We took a particular instance and verified that the data in `e` was live throughout much of the request lifecycle, and that `e->data` and `e->type->destroy` could be corrupted immediately after initialization (as long as `e->length` was also corrupted to 0) without causing a crash before a call to `apr_bucket_delete` was made. In particular the function which makes the call to this ACICS is `ap_get_brigade`.

Patterns like this occur even more frequently in BIND, where many structs are effectively objects with a "methods" field; an ex-

```
#define apr_bucket_destroy(e)
do {
  (e)->type->destroy((e)->data);
  (e)->free(e);
} while (0)
```

Figure 10: *bucket_brigade_destroy* macro definition in APR-util

```
result = xfr->stream->methods->next(xfr->stream);
```

Figure 11: Example call from BIND xfrout.c

ample is seen in Figure 11 displaying the same pattern observed in an APR-util function.

Clearly, if the bucket brigade struct or the XFR struct were implemented as C++ classes, they would need to be protected by a scheme such as v-table verification [55]. Fine-grained CFI schemes which can make strong guarantees about typical C programs will fail to account for cases such as this or other higher-level patterns implementing, without language support, object-oriented techniques, or closures on top of the C runtime. Programs that employ these patterns blur the traditional distinction between "data flow" and "control flow" attacks and present a significant challenge for static analysis, let alone CFI.

6.3.2 Security Implications of RTC

Even assuming a correct, compatible, and performant type-checking runtime, in-graph control flow attacks will still be possible as many program functions alias in signature. To evaluate exactly how much flexibility would be left, we wrote a tool that uses the libclang [2] library from Clang 3.6.0 to identify function signatures and indirect call signatures. We ran this tool against several popular server-side applications, reporting the number of matching function signatures for indirect call sites and function declarations. Note that the numbers include the linked headers from the standard libraries and the application libraries, as their functions are viable potential targets.

Number of unique...	HTTPD + APR	BIND	vsftpd	Nginx
...function names	9158	10125	1421	2344
...function signatures	5307	5729	730	1135
...indirect call signatures	117	135	5	3
...aliased signatures	81	27	5	2
...aliased functions	553	328	68	119

Table 8: Matching ICS & Function Signatures

The versions of the applications shown in Table 8 are Apache HTTPD 2.4.12, APR 1.5.1, BIND 9.10.2, vsftpd 3.0.2, and Nginx 1.7.11. The number of unique aliased signatures refers to the number of signatures found which matched both an indirect call and a function declaration. At a minimum, this number should be equal to the number of indirect call sites in the program. The last entry, the number of unique aliased functions, refers to the total number of functions (rather than function signatures) that could be potential targets of these indirect call sites. Thus, if at least one ACICS can be found for each aliased signature, the number of aliased functions is the number of functions that can be targeted while staying within the type system. Of course, a number of these are valid targets, but without programmer annotations the true number of unintentionally aliased pairs cannot be determined.

There are many pairings with few call sites and many target functions. For example, in BIND the signature *void()* matches 3 indirect calls and 252 target functions. In Apache, the signature `void (apr_pool_t *, void *, void *)` matches 2 indirect calls and 59 target functions, and `apr_status_t(void *)` matches 3 indirect call sites and 93 target functions. Even the relatively small vsftpd has 48 matches for its one indirect call with `void()`, and 11 for its one call with signature `void(void *)`.

The principled solution to reducing this problem entirely would be explicit programmer annotations for any aliasing function signature. However, the effort required to annotate all programs at this level of detailed would be immense.

7. RELATED WORK

Control Flow Bending (CFB) [15] also demonstrates, independently and concurrently with our work, attacks against fine-grained CFI. To perform their proof-of-concept attacks, Control Flow Bending introduces the notion of printf-oriented programming, a form of ACICS gadgets, that can be used to perform Turing-complete computation. CFB assumes a fully-precise CFG, which we show is undecidable. CFB relies on manual analysis for attack construction and is only able to achieve remote code execution in one of their six benchmarks. Moreover, printf-oriented programming is only applicable to older versions glibc. In the newer versions, the `%n` protection prevents the printf-oriented programming attack [11]. In contrast, Control Jujutsu introduces a framework (policies and tools) that enable automatic attack construction. CFB and Control Jujutsu demonstrate attacks against fine-grained CFI are possible in theory and in practice.

The Out of Control work by Göktas *et al.* [22] shows that coarse-grained implementations of CFI (with 2 or 3 tags) can be bypassed. In contrast, we show that even the fine-grained implementation of CFI with unlimited number of tags and a shadow stack using the state-of-the-art context- and field- sensitive static analysis is bypassable by a motivated attacker. Moreover, by studying the inherent limitations of scalable static analysis techniques, we show that attacks such as Control Jujutsu are hard to prevent using CFI.

Counterfeit Object Oriented-Programming (COOP) [46] is another recent attack on modern CFI defenses. COOP focuses exclusively on C++ by showing that protecting v-table pointers in large C++ programs is insufficient. Their work, like ours, focuses on showing certain design patterns that are common in sufficiently large or complex applications and are not accounted for in the design of CFI defenses. There may be some extensions of the COOP approach to C programs (particularly ones making heavy use of the patterns we described early); we leave this exploration to future work.

On the defense side, a number of recent fine-grained CFI techniques have been proposed in the literature. Forward-edge CFI [55] enforces a fine-grained CFI on forward-edge control transfers (i.e. indirect calls, but not returns). Cryptographically enforced CFI [34] enforces another form of fine-grained CFI by adding message authentication code (MAC) to control flow elements which prevents the usage of unintended control transfers in the CFG. Opaque CFI (OCFI) [36] enforces a fine-grained CFI by transforming the problem of branch target check to bounds checking (possible base and bound of allowed control transfers). Moreover, it prevents attacks on unintended CFG edges by applying code randomization. The authors of OCFI mention that it achieves resilience against information leakage (a.k.a. memory disclosure) attacks [47, 52] because the attacker can only learn about *intended* edges in such attacks, and not the *unintended* ones which were used in previous attacks against coarse-grained CFI [22]. Our attack shows that just the intended edges are enough for a successful attack.

Coarse-grained CFI efforts include the original CFI implementation [4], CCFIR [61], and Bin-CFI [62] all of which are bypassed by the Out of Control attack.

Software Fault Isolation (SFI) and SFI-like techniques also implement CFI at various granularities. Native Client [8, 60], XFI [20], and WIT [5] are some of those examples.

Other randomization-based [10, 27, 28, 57] and enforcement-based defenses [9, 58] against memory corruption attacks have been proposed and studied in the literature. Due to space limitations, we do not discuss them in detail here. Interested readers can refer to the surveys in the literature for a list of these defenses [39, 53].

8. CONCLUSION

We present a new attack, Control Jujutsu, that exploits the imprecision of scalable pointer analysis to bypass fine-grained enforcement of CFI (forward and backward edge). The attack uses a new "gadget" class, Argument Corruptible Indirect Call Site (ACICS), that can hijack control flow to achieve remote code execution while still respecting control flow graphs generated using context- and field-sensitive pointer analysis.

We show that preventing Control Jujutsu by using more precise pointer analysis algorithms is difficult for real-world applications. In detail, we show that code design patterns for standard software engineering practices such as extensibility, maintainability, and modularity make precise CFG construction difficult.

Our results provide additional evidence that techniques that trade off memory safety (security) for performance are vulnerable to motivated attackers. This highlights the need for fundamental memory protection techniques such as complete memory safety and indicates that the true cost of memory protection is higher than what is typically perceived.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful feedback and our shepherd Hovav Shacham for his help with the camera ready version of the paper. We also thank Deokhwan Kim, Vladimir Kiriansky, and William Streilein for their support, feedback and suggestions for improving this paper. This research was supported by DARPA (Grant FA8650-11-C-7192) and the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002.

10. REFERENCES

[1] Apache Portable Runtime Project. https://apr.apache.org/.
[2] Clang. http://clang.llvm.org/.
[3] The LLVM Compiler Infrastructure. http://llvm.org/.
[4] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proc. of ACM CCS* (2005).
[5] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with wit. In *Proc. of IEEE S&P* (2008).
[6] ANDERSEN, L. O. Program analysis and specialization for the c programming language. Tech. rep., 1994.
[7] ANDERSON, J. P. Computer security technology planning study. volume 2. Tech. rep., DTIC Document, 1972.
[8] ANSEL, J., MARCHENKO, P., ERLINGSSON, Ú., TAYLOR, E., CHEN, B., SCHUFF, D. L., SEHR, D., BIFFLE, C. L., AND YEE, B. Language-independent sandboxing of just-in-time compilation and self-modifying code.
[9] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proc. of ACM CCS* (2014).
[10] BIGELOW, D., HOBSON, T., RUDD, R., STREILEIN, W., AND OKHRAVI, H. Timely rerandomization for mitigating memory disclosures. In *Proc. of ACM CCS* (2015).
[11] BILAR, D. https://twitter.com/grsecurity/status/631670791445217280. In *GRSecurity's Twitter Feed* (2015).
[12] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIERES, D., AND BONEH, D. Hacking blind. In *Proc. of IEEE S&P* (2014).
[13] BLETSCH, T., JIANG, X., FREEH, V., AND LIANG, Z. Jump-oriented programming: A new class of code-reuse attack. In *Proc. of ACM CCS* (2011).
[14] BURROWS, M., FREUND, S. N., AND WIENER, J. L. Run-time type checking for binary programs. In *Proc. of the CC* (2003).
[15] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security* (2015).
[16] CARLINI, N., AND WAGNER, D. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium* (2014).
[17] CHEN, X., CASELDEN, D., AND SCOTT, M. New zero-day exploit targeting internet explorer versions 9 through 11 identified in targeted attacks, 2014.
[18] CHISNALL, D., ROTHWELL, C., WATSON, R. N., WOODRUFF, J., VADERA, M., MOORE, S. W., ROE, M., DAVIS, B., AND NEUMANN, P. G. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. *SIGPLAN Not.* (2015).
[19] COWAN, C., BEATTIE, S., DAY, R. F., PU, C., WAGLE, P., AND WALTHINSEN, E. Protecting systems from stack smashing attacks with stackguard. In *Linux Expo* (1999), Citeseer.
[20] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. Xfi: Software guards for system address spaces. In *Proc. of the OSDI* (2006).
[21] EVANS, I., FINGERET, S., GONZÁLEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. of IEEE S&P* (2015).
[22] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *Proc. of IEEE S&P* (2014).
[23] HARDEKOPF, B., AND LIN, C. Semi-sparse flow-sensitive pointer analysis. In *Proc. of POPL* (2009).
[24] HASTI, R., AND HORWITZ, S. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proc. of PLDI* (1998).
[25] HIND, M. Pointer analysis: Haven't we solved this problem yet? In *Proc. of PASTE* (2001).
[26] HIND, M., BURKE, M., CARINI, P., AND CHOI, J.-D. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.* (1999).
[27] HISER, J., NGUYEN, A., CO, M., HALL, M., AND DAVIDSON, J. Ilr: Where'd my gadgets go. In *Proc. of IEEE S&P* (2012).
[28] HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. librando: Transparent code randomization for just-in-time compilers. In *Proc. of ACM CCS* (2013).
[29] INTEL. Introduction to intel memory protection extensions, 2013.

[30] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of c. In *USENIX Technical Conference* (2002).

[31] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity.

[32] KWON, A., DHAWAN, U., SMITH, J., KNIGHT, T., AND DEHON, A. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proc. of ACM CCS* (2013).

[33] LATTNER, C., LENHARTH, A., AND ADVE, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. of PLDI* (2007).

[34] MASHTIZADEH, A. J., BITTAU, A., MAZIERES, D., AND BONEH, D. Cryptographically enforced control flow integrity.

[35] MICROSOFT. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. Online, September 2006.

[36] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K., AND FRANZ, M. Opaque control-flow integrity. In *Proc. of NDSS* (2015).

[37] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices* (2010).

[38] NECULA, G. C., MCPEAK, S., AND WEIMER, W. Ccured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices 37*, 1 (2002), 128–139.

[39] OKHRAVI, H., HOBSON, T., BIGELOW, D., AND STREILEIN, W. Finding focus in the blur of moving-target techniques. *IEEE Security & Privacy 12*, 2 (Mar 2014).

[40] ONE, A. Smashing the stack for fun and profit. *Phrack magazine 7*, 49 (1996), 14–16.

[41] OPENBSD. Openbsd 3.3, 2003.

[42] PEARCE, D. J., KELLY, P. H., AND HANKIN, C. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst. 30*, 1 (Nov. 2007).

[43] RAMALINGAM, G. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16*, 5 (Sept. 1994), 1467–1471.

[44] REPS, T. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst. 22*, 1 (Jan. 2000), 162–186.

[45] RUGINA, R., AND RINARD, M. Pointer analysis for multithreaded programs. In *Proc. of PLDI* (1999).

[46] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming. In *Proc. of IEEE S&P* (2015).

[47] SEIBERT, J., OKHRAVI, H., AND SODERSTROM, E. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proc. of ACM CCS* (2014).

[48] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *USENIX Technical Conference* (2012).

[49] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc.of ACM CCS* (2007).

[50] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. of IEEE S&P* (2013).

[51] SRIDHARAN, M., AND BODÍK, R. Refinement-based context-sensitive points-to analysis for java. In *Proc. of PLDI*.

[52] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *Proc. of EuroSec* (2009).

[53] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *Proc. of IEEE S&P* (2013).

[54] THE PAX TEAM. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

[55] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium* (2014).

[56] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V., AND NING, P. On the expressiveness of return-into-libc attacks. In *Proc. of RAID'11* (2011).

[57] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of ACM CCS* (2012).

[58] WATSON, R. N., WOODRUFF, J., NEUMANN, P. G., MOORE, S. W., ANDERSON, J., CHISNALL, D., DAVE, N., DAVIS, B., LAURIE, B., MURDOCH, S. J., NORTON, R., ROE, M., SON, S., VADERA, M., AND GUDKA, K. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Proc. of IEEE S&P* (2015).

[59] WHALEY, J., AND LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of PLDI* (2004).

[60] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. of IEEE S&P* (2009).

[61] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Proc. of IEEE S&P* (2013).

[62] ZHANG, M., AND SEKAR, R. Control flow integrity for cots binaries. In *USENIX Security* (2013), pp. 337–352.