Practical Data-Only Attack Generation

Brian Johannesmeyer Asia Slowinska Herbert Bos Cristiano Giuffrida

Vrije Universiteit Amsterdam

Abstract

As control-flow hijacking is getting harder due to increasingly sophisticated CFI solutions, recent work has instead focused on automatically building data-only attacks, typically using symbolic execution, simplifying assumptions that do not always match the attacker's goals, manual gadget chaining, or all of the above. As a result, the practical adoption of such methods is minimal. In this work, we abstract away unnecessary complexities and instead use a lightweight approach that targets the vulnerabilities that are both the most tractable for analysis, and the most promising for an attacker.

In particular, we present EINSTEIN, a data-only attack exploitation pipeline that uses dynamic taint analysis policies to: (i) scan for chains of vulnerable system calls (e.g., to execute code or corrupt the filesystem), and (ii) generate exploits for those that take unmodified attacker data as input. EINSTEIN discovers thousands of vulnerable syscalls in common server applications—well beyond the reach of existing approaches. Moreover, using `nginx` as a case study, we use EINSTEIN to generate 944 exploits, and we discuss two such exploits that bypass state-of-the-art mitigations.

1 Introduction

"Everything should be made as simple as possible, but not simpler." – attributed to Albert Einstein

Since control-flow hijacking attacks came to the fore over three decades ago [83], attacks and defenses have battled over "control" of a program's control flow. Such attacks follow the general steps listed in Figure 1a: they overwrite certain *control data* of a program; force it to execute attacker-specified code snippets, i.e., *gadgets*; and may even chain enough of these gadgets together to achieve arbitrary computation, i.e., *Turing completeness* [20, 21, 25, 51, 73, 79]. This escalated into an arms race—between attacks exploiting some control flow, and defenses restricting that control flow—until the wide deployment of control-flow integrity (CFI) [11, 88, 89, 91, 98], which made exploitation of a program's control flow increasingly difficult.

The promise of data-only attacks. In response, attackers turned to data-only attacks, which promised to exploit a program's (massive) set of data flows (i.e., any program data being used in any operation). Yet, although these attacks technically exploited *non-control data*, many of them still relied on exploiting control-*adjacent* data (e.g., branch conditions) to perform "legal" control-flow hijacking. In effect, they were still exploiting the (tiny) residual attack surface of a program's control flow. Hence, to identify data-only attacks from such a small attack surface, these approaches employed heavyweight analyses (e.g., symbolic execution) to reason about numerous complex constraints.

In reality, however, it is extremely difficult, if not impossible, to solve all constraints in bounded time for a sufficiently complex program. The problem space—i.e., overwriting *any* possible data to *any* possible value, using it in *any* potential gadget, from *every* possible program point, to achieve *arbitrary* computation—is too large. Hence, in order to scale, these approaches are forced to make several simplifying assumptions, e.g., that: (1) an attack must only overwrite specific parts of memory, or (2) an attacker must use another exploit to reach the gadget's entrypoint, or (3) an attacker must manually chain together gadgets to build an exploit. While these assumptions make exploit generation a tractable problem, they unfortunately either dramatically limit the scope of the resulting exploits, or leave parts of it to future or manual work. In short, the overarching complexity of these approaches prevents them from achieving all the goals listed in Figure 1b, thereby limiting their practical adoption.

A lightweight approach. In this paper, we investigate a novel lightweight technique to build data-only attacks. Our approach relies on four key insights that dramatically reduce the problem space. First, not unlike Newton's approach in the related domain of control-flow hijacking [90], we observe that dynamic taint analysis simplifies away many complex constraints for data-only attacks. In other words, rather than trying to reason about vast swathes of unknowns, we can simply reason about the concrete data used in a concrete

Step 1:	Exploit a *memory write* vulnerability.
Step 2:	Steer the execution to a gadget's *entry point*.
Step 3:	Execute some payload via a *gadget chain*.

(a) Steps in any non-control (or control) data attack.

Goal 1:	Automatically model an arbitrary *memory write* vulnerability.
Goal 2:	Automatically identify an arbitrary *gadget entry point*.
Goal 3:	Automatically produce a full *gadget chain*.

(b) Goals for automating data-only attacks.

Figure 1: Data-only attacks: steps and goals.

program execution. Second, we observe that expressiveness (and especially Turing completeness) is not necessary. Rather, attackers simply want to achieve specific goals, such as running code via `execve`, or writing to a file via `write`. Third, we observe that manipulating a program's intended path is not necessary. In reality, programs will invoke `execve`, `write` or other interesting system calls all by themselves, without any additional interference from an attacker. Finally, we observe that after initialization, a program treats many types of data as immutable, and hence, that data has few (if any) constraints. By targeting these straightforward "identity" dataflows, we can corrupt data (e.g., a filename) at one program point and expect it to remain unchanged all the way to another program point (e.g., the filename getting passed to `execve`)—all the while without performing any heavyweight constraint solving. Using these insights, we make the problem tractable, even for lightweight analysis.

We present Einstein, a data-only attack exploitation pipeline. Einstein uses custom taint policies (rather than constraint solving) to identify attacker-controllable syscalls (rather than Turing completeness) along a program's (already valid) runtime path, generating exploits for syscall arguments that have an identity dataflow (rather than a complex dataflow) from attacker data. Given a vulnerable target program as input, it instruments the code to model and track each step of an attack, allowing it to identify gadget chains at runtime.

Specifically, we first taint any data that could be overwritten by an attacker with a unique color. Then, we track the flow of taint into security-sensitive system calls [30, 34, 42, 66, 89]. Moreover, we track the flow of taint into certain library state that is shared across system calls. If this state can be controlled by one system call, and used by another, we have evidence that an attacker can exploit it to chain together two otherwise safe system calls in a data-only attack. Finally, we generate exploits that corrupt the identified syscall arguments with attacker data, and confirm the exploits' effectiveness by running them on the target application.

Einstein's low complexity approach allows it to identify thousands of vulnerable gadgets. Taking the popular web server `nginx` as a case study, we use Einstein to generate 944 exploits, including 1 Code-Execution primitive, 17 Write-What-Where primitives, and 41 Send-What-Where primitives. Moreover, many of these exploits bypass state-of-the-art mitigations, e.g., syscall filtering [36, 37, 46, 69] and selective DFI [46, 81].

As case studies, we discuss the Code-Execution exploit, as well as one of the Write-What-Where exploits. The former demonstrates the rich attack surface that data-only approaches offer, with plenty of low-hanging fruit available to an attacker. It exploits an `execve` syscall that takes its `pathname` and `argv` parameters directly from a global variable, allowing an attacker to trivially execute arbitrary code. The latter demonstrates the power of syscall chaining, which further expands the available attack surface. It exploits the `pathname` parameter of an `open` syscall and the `fd` and `buf` parameters of an unrelated `write` syscall, allowing an attacker to corrupt the server's filesystem.

Contributions. We make the following contributions:

- We present a practical approach to building data-only attacks in complex programs.
- We develop Einstein, an open-source[1] data-only exploitation pipeline.
- We evaluate Einstein on popular server applications to identify thousands of previously unknown vulnerable syscalls and use `nginx` as a case study to generate hundreds of working exploits.

2 Background & Related Work

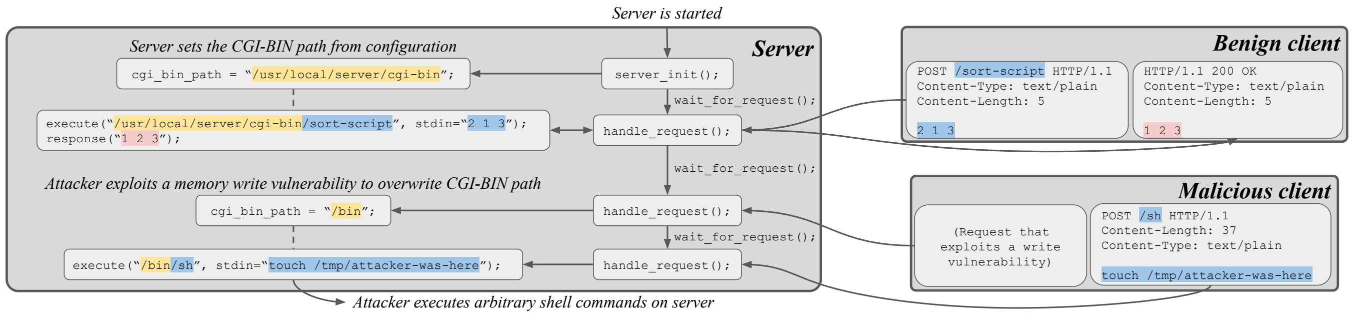
2.1 Data-Only Attacks & Defenses

Young and McHugh presented one of the first examples of a data-only attack in 1987 [97], and subsequent work suggested their viability in practice [31, 86, 93]. Then, Chen et al. presented the first extensive look of such attacks on real-world programs in 2005 [27].

Running example. Figure 2 outlines one of the classic data-only attacks described in the literature [27]. In this example, the server uses the `sort-script` program to sort numbers specified by a client. In the benign case, a client first sends a `POST /sort-script` request to the server with the unsorted numbers in the request body, which the server feeds to `/sort-script` in the the CGI-BIN directory. Finally, the program returns the sorted numbers to the server, and the server passes them on to the client. However, a memory write vulnerability in the Null HTTPD web server allows attackers to overwrite the CGI-BIN path, for instance by setting it to `/bin`. When the attacker now sends a `POST /sh` request to the server, it will execute the shell commands contained in the request body.

Practical vs. comprehensive defenses. Mitigations for these attacks generally attempt to prevent one of two operations: (1) the *definition* (i.e., *def*) of malicious data, i.e., the memory

[1]Einstein is available at https://github.com/vusec/einstein.


Server is started
Server
Server sets the CGI-BIN path from configuration
Attacker exploits a memory write vulnerability to overwrite CGI-BIN path
Attacker executes arbitrary shell commands on server
Benign client
Malicious client
(Request that exploits a write vulnerability)

```
cgi_bin_path = "/usr/local/server/cgi-bin";
server_init();
wait_for_request();
execute("/usr/local/server/cgi-bin/sort-script", stdin="2 1 3");
response("1 2 3");
handle_request();
wait_for_request();
cgi_bin_path = "/bin";
handle_request();
wait_for_request();
execute("/bin/sh", stdin="touch /tmp/attacker-was-here");
handle_request();
```

```
POST /sort-script HTTP/1.1
Content-Type: text/plain
Content-Length: 5

2 1 3
```

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 5

1 2 3
```

```
POST /sh HTTP/1.1
Content-Length: 37
Content-Type: text/plain

touch /tmp/attacker-was-here
```

Figure 2: The data-only attack presented by Chen et al. [27] (with minor modifications for clarity). Despite its discovery almost two decades ago, EINSTEIN still identifies similar gadgets that exploit the CGI-BIN path of popular web servers.

write vulnerability; or (2) the *use* of malicious data, e.g., passing malicious data to `execve`. Unfortunately, *comprehensive* solutions to prevent the *def* of malicious data (e.g., memory safety [13, 14, 23, 47, 58, 59, 96], data space randomization (DSR) [16, 18, 22, 70]) or to prevent its *use* (e.g., dataflow integrity (DFI) [24, 53, 82]) either incur poor performance or require onerous changes in the software or hardware, thereby limiting their adoption in practice. On the other hand, *practical* measures against the *def* of malicious data (e.g., memory error scanning [12, 38, 77, 78, 85], selective DSR [64, 65]) or against its *use* (e.g., selective DFI [46, 81], syscall filtering [36, 37, 46, 69]) are non-comprehensive, because they leave a significant portion of the attack surface vulnerable to attacks.[2]

2.2 Automated Data-Only Attacks

Despite the discovery of data-only attacks such as the one in Figure 2 almost two decades ago and the lack of practical, comprehensive mitigations, automatic solutions to building attacks are still limited in a few ways: they are limited in scope, require significant manual effort, or make unrealistic assumptions. These limitations often relate to the inherent complexity and poor scalability of the concolic/symbolic execution that underlies most of these solutions, but sometimes their objectives are also different. For instance, some approaches strive for Turing completeness, which is interesting from an academic viewpoint, but not so relevant for attackers. In the remainder of this section, we break down various ways in which previous approaches fall short (see also Table 1).

Vulnerability-dependent analysis. While preventing the use of malicious data requires a comprehensive attack surface analysis: the identification (and elimination) of *any* dangerous data flow, regardless of its origin, previous approaches [41, 43, 68] have a narrower scope, limiting themselves to specific memory write vulnerabilities rather than generic primitives. In other words, the attacks they generate are *only applicable* to the *specific vulnerability* supplied by the user, including the specific addresses and values it may write. A partial exception is Limbo [75], which models arbitrary stack overflows, although that is still a long way from general memory corruption. In contrast, EINSTEIN starts its analysis from *generic attacker capabilities* (e.g., an arbitrary read/write primitive), regardless of the specific vulnerability.

Predetermined gadget entry point. Next, previous approaches may require the user to predefine the gadget entry point [45, 68]. In other words, in order to build attacks, they need the user to *already know how to exploit the program* in a way that steers its execution to the first gadget. This is the typical model of any "weird compiler" whose goal is to chain together Turing-complete gadget chains (e.g., ROP compilers [74], which assume the user can already exploit one return instruction), and it is also the model for data-only gadget compilers [45, 68]. However, identifying an entry point can be a non-trivial task, because it often requires a user to develop a separate exploit that just forces the program's execution to reach the first gadget (e.g., by "legally" diverting control flow). Previous approaches may *partially* model the entry point by either being incomplete (e.g., due to the poor coverage of concolic execution [41, 75]) or unsound (e.g., by assuming any loop executed after a memory error could be a gadget dispatcher [43]).

Manual gadget chaining. Finally, previous approaches may require the user to manually chain gadgets [43]. However, *chaining is often a non-trivial task*, as CFI limits which gadgets may be chained together and gadgets may be "volatile" (i.e., selecting one gadget in a chain may rule out the selection of other gadgets in that chain [45]). Several previous approaches do not handle chaining at all, while all others do so only partially (e.g., due to concolic execution's poor coverage [41, 75], or the problem being NP-hard [45]).

3 Threat Model

We assume mostly the same threat model as previous work [41, 43, 45, 68, 75]. In particular, we consider a target program that: (1) has a memory corruption vulnerability [1, 3–6]; and (2) is protected with DEP [15], ASLR [67], state-of-the-art control-flow hijacking defenses (e.g., CFI [11, 88, 89, 91, 98]), and strong stack protections (e.g., perfect stack canaries, shadow stacks [32]). This differs slightly from pre-

[2] We refer the interested reader to previous surveys of data-only attacks [28, 29] and memory corruption defenses [87].

Table 1: Overview of approaches to building data-only attacks and how they model the steps in Figure 1a. Specifically, whether they: can model an attack step (●), can *partially* model an attack step (◐), or require an attack step to be defined by the user (○).

Approach		Design		Models this attack step?			Is practical?	
Name	Year	Type of dataflow analysis	Goal	Memory write vulnerability	Gadget entry	Gadget chaining	Models all primitives	Models primitive completely
FlowStitch [41]	2015	Concolic execution	Attacker-centric	○	◐	◐	✗	✗
MinDOP [43]	2016	Static taint analysis	Turing-complete	○	◐	○	✗	✗
BOPC [45]	2018	Symbolic execution	Turing-complete	●	○	◐	✗	✓
Steroids [68]	2019	Static constraint solving	Turing-complete	○	○	●	✗	✓
Limbo [75]	2020	Concolic execution	Attacker-centric	◐	◐	◐	✓	✗
EINSTEIN	-	Dynamic taint analysis	Attacker-centric	●	●	●	✓	✓

vious work [41, 43, 45, 68, 75], which do not assume strong stack protections. For simplicity, we focus specifically on popular server programs, similar to much prior work in the area [19, 54, 61, 62, 76, 89–91].

Moreover, we assume that an attacker: (1) has access to a binary equivalent of the one deployed by their prospective victim; (2) can legitimately interact with the target program (e.g., by sending requests to a target web server); (3) can bypass ASLR via an information leak [35, 39, 44, 72, 80]; (4) can exploit the memory corruption vulnerability for an arbitrary write primitive from a quiescent program state, as in previous work [62, 90]; and (5) aims to force the target program to invoke system calls with attacker-controlled parameters [30, 34, 42, 66, 89]. This differs from previous work, which assumes: (1) an attacker can carry out the arbitrary write primitive from an *arbitrary* program state [45], rather than a *quiescent* state; and (2) an attacker that aims for *Turing completeness* [43, 45, 68], rather than an attacker that aims for *security-sensitive controllable system calls* (which we introduce later, see Table 2).

4 Overview of EINSTEIN in Action

Before we discuss our design in detail, we walk through EINSTEIN's operation with our running example. We consider a target program, e.g., the web server, to have three application-dependent phases of execution: (i) an *initialization phase*, where the program starts and initializes long-lived data, e.g., configuration data; (ii) a *quiescent phase*, where the program waits for user interaction, e.g., by idly waiting for new client requests, at which point an attacker may exploit the memory write vulnerability; and (iii) a *processing phase*, where the program handles user interaction, e.g., by invoking `execve`.

To guide our explanation of EINSTEIN's operation, we refer the reader to Figure 3, which shows the main components at the top and the main analysis steps at the bottom. Next, we explain how EINSTEIN identifies (Steps 1–4) and builds (Steps 5–6) the attack of Figure 2. We defer the more complex aspects of EINSTEIN, such as the exact taint policies and chaining, until later.

Step 1. We use a *program driver* (e.g., a test suite) to run the victim program with EINSTEIN instrumentation. As in previous work [62, 90], EINSTEIN begins its analysis with the program in a quiescent state, by which time, all initialization (e.g., of `cgi_bin_path`), has completed.

Step 2. EINSTEIN then models an arbitrary memory write vulnerability by tainting all data that could be affected by it (again, including `cgi_bin_path` string), thereby fulfilling **Goal 1** of Figure 1b. This is the first of our *taint policies*. Additionally, it records the tainted data in a memory snapshot.

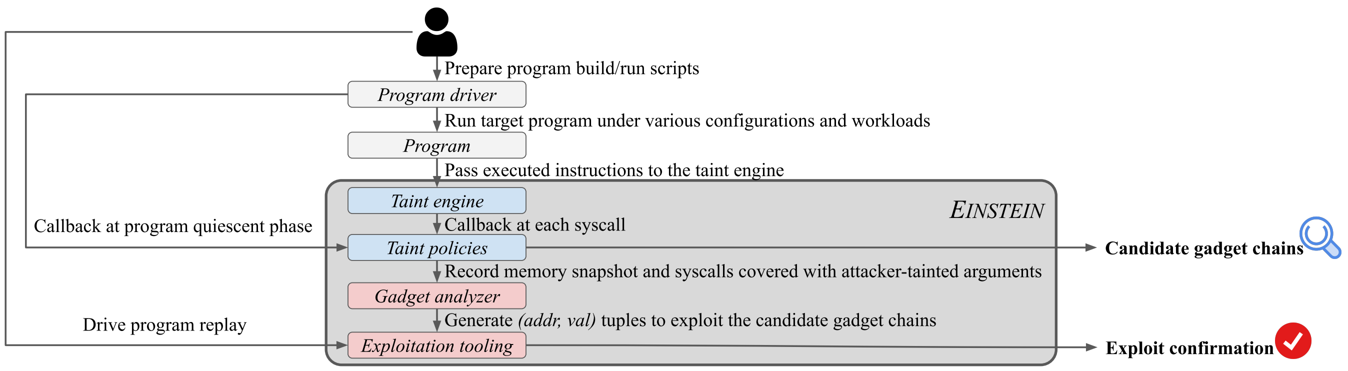
Step 3. To uncover gadgets, the driver sends standard requests to the server (e.g., the `POST /sort-script` request from the example), while the taint engine propagates the flow of attacker data through the target server's execution.

Step 4. As the server handles the example `POST` request, it invokes `execve` with attacker-tainted arguments. Recognizing the system call as sensitive, EINSTEIN's second *taint policy* identifies it as a *candidate gadget*, fulfilling **Goal 2**. Additionally, it records information about the syscall, such as its arguments and their taintedness.

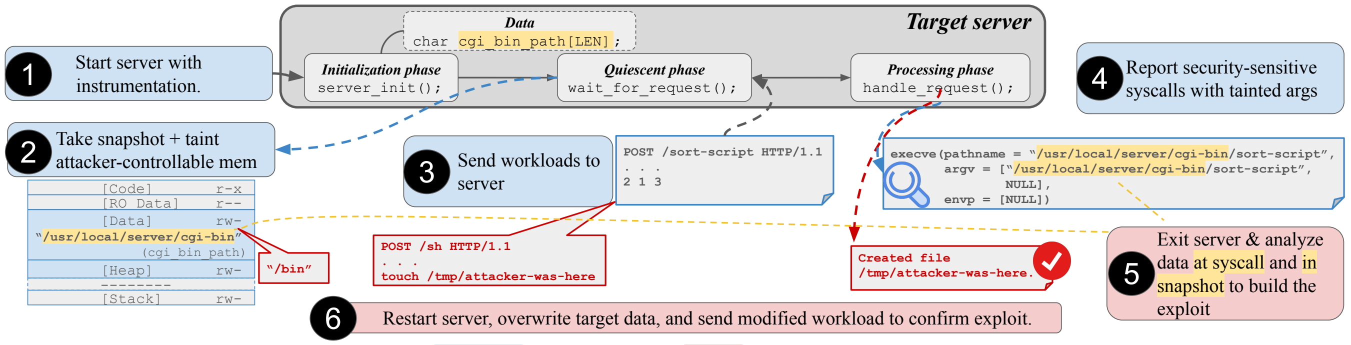
Step 5. Having identified the candidate gadget, it now tries to build an exploit. First, EINSTEIN's *gadget analyzer* determines that parts of `execve`'s `pathname` and `argv` arguments are tainted with a color that corresponds to `cgi_bin_path`. Upon further inspection, it finds that they are in fact identical to `cgi_bin_path`. We refer to this kind of straightforward dataflow as an *identity dataflow*. EINSTEIN builds a candidate exploit by generating $(addr, val)$ pairs to overwrite the target data from `"/usr/local/server/cgi-bin"` to `"/bin"` (**Goal 3**).

Step 6. To confirm its effectiveness, EINSTEIN's *exploitation tooling* restarts the server, overwrites the target data defined by the $(addr, val)$ pairs, and sends the workload to exploit the gadget—in this case `POST /sh` with a shell command to create a file in the /tmp directory. If the file is created, EINSTEIN confirms the exploit to be successful.

For the running example, a single-blow overwrite of the arguments of a single system call is sufficient for exploitation.


Prepare program build/run scripts
Program driver
Run target program under various configurations and workloads
Program
Pass executed instructions to the taint engine
EINSTEIN
Taint engine
Callback at each syscall
Callback at program quiescent phase
Taint policies
Candidate gadget chains
Record memory snapshot and syscalls covered with attacker-tainted arguments
Gadget analyzer
Generate (addr, val) tuples to exploit the candidate gadget chains
Drive program replay
Exploitation tooling
Exploit confirmation

(a) Architecture.


Target server
Data
char cgi_bin_path[LEN];
1 Start server with instrumentation.
Initialization phase
server_init();
Quiescent phase
wait_for_request();
Processing phase
handle_request();
4 Report security-sensitive syscalls with tainted args
2 Take snapshot + taint attacker-controllable mem
[Code] r-x
[RO Data] r--
[Data] rw-
"/usr/local/server/cgi-bin"
(cgi_bin_path)
[Heap] rw-

[Stack] rw-
"/bin"
3 Send workloads to server
POST /sort-script HTTP/1.1
. . .
2 1 3
execve(pathname = "/usr/local/server/cgi-bin/sort-script",
argv = ["/usr/local/server/cgi-bin/sort-script",
NULL],
envp = [NULL])
POST /sh HTTP/1.1
. . .
touch /tmp/attacker-was-here
Created file
/tmp/attacker-was-here.
5 Exit server & analyze data at syscall and in snapshot to build the exploit
6 Restart server, overwrite target data, and send modified workload to confirm exploit.

(b) Operation: EINSTEIN identifies (Steps 1–4) and builds (Steps 5–6) the example attack in Figure 2.

Figure 3: Overview of EINSTEIN.

While simple, such cases are still common (Section 7). On the other hand, we will show that EINSTEIN supports system call chaining also when we discuss the complete set of EINSTEIN's taint policies. We demonstrate the usefulness of chaining in our second case study (Section 8).

5 Design

In this section, we discuss EINSTEIN's design by introducing each of the components in Figure 3a in some detail.

5.1 Taint Engine

To track attacker-controllable data at runtime, a dynamic taint analysis (DTA) engine models the flow of data from each executed instruction's inputs to its outputs. To do this, DTA engines make use of the following constructs (which may have different names depending on the engine): (1) a *color* is some kind of program metadata, which in our case, is an address that can be overwritten by the attacker; (2) a *tag* (also called a *tagset*) is the set of colors corresponding to some program data (e.g., a memory location), which in our case, indicates that some data is attacker-controllable; and (3) the *tagmap* contains the tags for all program data. This is a fairly typical design of DTA [26, 48, 60, 71, 84, 90], and indeed our design is inspired by a DTA engine [90] that was used by previous work to identify control data attacks[3].

However, our use case, i.e., non-control data attacks, poses two unique challenges, which existing DTA engines cannot solve. First, because all non-control data is a potential attack vector, we need an approach that can uniquely track an *unbounded* number of taint colors over an *unbounded* amount of data. For instance, in our example attack, we need to know that, out of all the possible data that our exploit could overwrite, it should specifically overwrite the server's CGI-BIN path. Second, because programs typically invoke system calls at complex points (e.g., a time-sensitive network write, or a large file read), we need an approach that can *scale* to complex workloads covering complex features. For instance, in our example attack, we need to indeed be able to cover the codepath that handles the exploit's POST request. In contrast, we confirmed previous similar DTA engines cannot easily scale beyond a single default request to a server program, as the excessive slowdowns cause complex requests to time out [90]. Hence, it is around *scalability* and supporting *unbounded colors* for *unbounded data* that we design three fundamental

[3]Their approach, which was the successor to prior work named Galileo [79] (*"space"*), was named Newton [90] (*"absolute space and time"*). Since our approach generalizes the approach of Newton to non-control data attacks, we name it EINSTEIN (*"spacetime"*).

aspects of our DTA engine, in particular: (1) how it accesses the tagmap, (2) how it initializes the tagmap, and (3) how it combines tags.

Accessing the tagmap. Every time a program accesses data (e.g., via a load or store), the DTA engine accesses its tag. The DTA engine typically does this by taking the program data's address and performing some mapping to locate the corresponding tag. Some DTA engines may be *slow but memory-efficient* by navigating through multiple layers of a dynamically managed data structure (e.g., page tables) on every access [9, 10, 49, 90]. Others may be *fast but memory-inefficient* by indexing into a statically managed data structure (e.g., shadow memory) on every access [8, 85].

Our approach takes the best of both worlds by combining a *fast* shadow memory organization with a *memory-efficient* design. Specifically, we instruct the linker to constrain the user address space to the top 4 GB and reserve the lower part for shadow memory. This organization yields (i) fast shadow memory-based mappings between addresses and corresponding tags (only 2 arithmetic and 1 masking operations needed) and (ii) 64-bit user pointers easily compressible to 32 bits, since the top 32 bits are always identical. Since we use pointers as taint colors, such pointer compression strategy [52] also translates to our (32-bit) colors, thereby reducing the overall size of the tagmap by 50%.

Initializing the tagmap. Before accessing a tainted entry in the tagmap, the DTA engine must first initialize the tagmap. Some DTA engines may be *fast but taint limited data* by starting with an untainted (i.e., zero-initialized) tagmap [8–10, 77, 85]. Such approaches take advantage of the fast hardware MMU, because any access to an uninitialized tag page generates a page fault, which then automatically maps a zero-initialized page into the tagmap. Other DTA engines may be *slow but taint unbounded data* by starting with a mostly-tainted tagmap [84, 90]. Such approaches require a slow software-based MMU to check every access, so that when an uninitialized tag page is identified, they can map a custom-initialized page into the tagmap.

Our approach takes the best of both worlds by combining a tagmap that can taint *unbounded data* while enabling *fast* access checks by the hardware MMU. In particular, we use Linux's `userfaultfd` feature to: (1) quickly identify an uninitialized tag page via a hardware page fault, and (2) handle the page fault in software, so our DTA engine can taint the entries of the tagmap that represent attacker-controllable data.

Combining tags. Every time a program combines data (e.g., via arithmetic), the DTA engine combines their tags. The DTA engine does this by performing a set union of the colors in one tag with the colors in another tag. Some DTA engines may aim to be *fast but support few colors* by representing a tag as a size-limited array, with an entry allocated for every possible color—e.g., an array containing either 1 color [10, 49, 85], 8 colors [8, 49], 256 colors [7, 9], etc. Other DTA engines may aim to be *slow but support unbounded colors* by representing a tag as an unbounded set, with an entry for each color added at runtime [84, 90].

Our approach takes the best of both worlds by using a tagset that *supports unbounded colors*, while limiting its runtime capacity in order to stay *fast*. In particular, our array-based tagset contains an entry for each color added at runtime, up to some limit N. If a tagset contains more than N colors at runtime, then we denote the tagset as *overfilled* and do not combine any more colors into it. We rely on the insight that if a tag contains numerous colors, then it likely represents complex data, with multiple sources (e.g., the output of a cryptographic hash), and hence, that data would be very challenging for an attacker to exploit than a tag with few colors. For our experiments, we select $N = 16$ as it produces few tagset overfills while maintaining an acceptable memory overhead.

5.2 Taint Policies

We design taint policies to model the steps of a data-only attack listed in Figure 1a. At a high level, we initially mark attacker-controllable regions as tainted, while EINSTEIN's DTA engine propagates the taint information to syscall callsites, and checks the tags of syscall arguments to identify those that may be exploitable.

5.2.1 Modeling an arbitrary memory write

We model the arbitrary memory write by tainting all memory that an attacker could overwrite. We discuss *which* data is attacker-controllable, and *what* we taint it with.

Because we consider an arbitrary memory write primitive, we taint data in *any writable memory segment*. However, since we focus on long-lived (and exclude short-lived) data corruption effected by the primitive, we *exclude the stack's segment*. (It is not a fundamental limitation, and our approach can easily accommodate tainting the long-lived data on the stack.) Moreover, to model the effect of long-lived data corruption, we taint memory when the program is in a quiescent state, as done in prior work [90].

For EINSTEIN's analysis it is vital to be able to precisely identify the origin of every tainted byte. To track memory dependencies at a byte granularity, we configure our DTA engine with a unique tag for every attacker-controllable byte: specifically, the compressed (32-bit) version of its memory address. To track the specific *value* that it originates from, we also record a memory snapshot tainting all memory. Hence, for any tainted tag, we have its origin: its $(addr, val)$ pair. The source of the taint is later a candidate location for the attacker to corrupt, control a syscall argument, and launch their attack.

5.2.2 Tracking dependencies

Traditional taint implementations [49] model taint strategies that track direct attacker-controlled memory dependencies (i.e., syscall argument X was read at tainted address Y), and ignore indirect ones (i.e., syscall argument X was read at address Y' using a tainted pointer). To support the latter, we additionally implemented load pointer propagation (i.e., taint every value read via a tainted pointer), allowing us to model attackers corrupting data pointers, array indexes, etc. to indirectly control syscalls arguments.

5.2.3 Modeling syscall exploitation

We model the next part of an attack, the execution of a gadget, with taint sinks that check whether syscall arguments are attacker-controllable. Specifically, we insert hooks before the invocation of the security-sensitive syscalls listed in Table 2. If we identify any attacker-tainted arguments, we generate a syscall report that includes: (1) info about the syscall, such as its name, arguments, arguments' taint, and backtrace; (2) info to distinguish it from other reports, such as the target process's PID and TID, and an incrementing report number; and (3) info to help reproduce the exploit.

5.2.4 Modeling syscall chaining

We model gadget chaining with a combination of taint sources and taint sinks. We note, however, that chaining may not even be required for many end-to-end attacks. Namely, because our gadgets are syscalls—rather than small snippets that e.g., perform arithmetic, or conditional branching. [20, 21, 25, 45, 51, 68, 73, 79]—one gadget may suffice for an entire attack. For instance, the example attack in Figure 2 only exploits *one* syscall, `execve`. However, if such a syscall is unavailable to the attacker (e.g., due to mitigations), syscall chaining greatly expands the set of available gadgets.

Chaining syscalls via file descriptors. Just like any other gadget compiler [45, 68, 74], we chain syscalls by chaining the data output by one gadget into the input of another gadget. The syscall interface may chain together many different types of data: process IDs (from `getpid` to `kill`), semaphore IDs (from `semget` to `semop`), and so on. EINSTEIN chains *file descriptors (FDs)*, as they are used by some of the security-sensitive syscalls that we target. An FD uniquely identities an open *I/O stream*, e.g., a file or a network socket. If an attack controls an I/O stream, then it controls *where* the target program sends and receives data. We focus on output streams as they allow attackers to effect changes, but our approach equally applies to input streams.

Table 2 lists the syscalls that EINSTEIN targets and that are relevant for chaining. In particular, some security-sensitive syscalls only operate on some types of FDs (e.g., `sendto` only takes a socket FD), so they can only be chained to specific FD-creating syscalls (e.g., `socket`). On the other hand, `write` can operate on multiple types of FDs, and hence, can be chained with multiple types of FD-creating syscalls.

I/O streams can be either *directly* or *indirectly* controllable by an attacker. First, an I/O stream is *directly* controllable if the attacker controls how it was created, i.e., the arguments to an FD-creating syscall, e.g., the `filename` argument to `open`. Second, an I/O stream is *indirectly* controllable if the attacker controls an FD argument to a security-sensitive syscall and can redirect it to a directly-controllable I/O stream. For example, by controlling the `fd` argument to `write`, the attacker can redirect it to the aforementioned directly-controllable file. Because an indirectly-controllable stream depends on the existence of a directly-controllable stream, we first explain how we track directly-controllable streams, then we explain how we identify directly- and indirectly-controllable accesses.

Tracking file descriptors. To identify directly-controllable output streams, we add taint sinks to FD-creating syscalls that check their arguments. First, our sinks determine whether the created stream can indeed be an *output* stream. For some syscalls, this is not necessary to check, but for others, this is specified by the arguments (e.g., `open`'s `flag` argument determines whether the file is writable). In such cases, we conclude that the syscall can create an output stream if: (1) the relevant arguments specify to create one, or (2) the relevant arguments are attacker-tainted, and hence, an attacker can coerce it into creating one. Second, our sinks determine whether an attacker can *directly control* "where" the output stream will go by checking if relevant arguments are attacker-tainted, e.g., `open`'s `pathname` or `connect`'s `addr` arguments.

Next, we need a way to track these directly-controllable streams. Namely, the kernel maintains an FD table, which maps FDs to their corresponding I/O streams. If attackers control a created stream, then they control its entry in the FD table. Hence, any future references to an attacker-controllable stream should be treated as tainted—regardless of whether the FD is tainted. To model this, we maintain our own *controllable FD table*, which mirrors the kernel's FD table, but with a few differences. Specifically, our table: (1) only creates entries for controllable output streams, (2) maps FDs to the report number of the syscall that created it, and (3) is consulted to taint any subsequent FD syscall argument that may refer it (whether directly or indirectly).

Identifying accesses to controllable file descriptors. The final question we reach in our design is: How do we identify accesses to attacker-controllable I/O streams? The naive approach would be to use the same taint sinks described in Section 5.2.3 to simply check whether FD arguments to security-sensitive syscalls are tainted. However, this is imprecise because it would: (1) fail to identify many controllable accesses, e.g., if the FD argument is *untainted*, but its I/O stream is *tainted*; and (2) falsely identify many non-controllable accesses as controllable, e.g., if the FD argument is *tainted*, but all open I/O streams are *untainted*. Hence, we modify our taint sinks for FD arguments to instead check our *controllable FD table*. In particular, we determine that a syscall's FD argument is: (1) directly-controllable if it exists in our controllable FD table; or (2) indirectly-controllable if it is tainted and our controllable FD table contains an FD of the same type (e.g., a socket FD, if the syscall requires a socket FD).

Table 2: Syscalls targeted by EINSTEIN. We target a similar set of security-sensitive syscalls as previous work [30, 34, 42, 66, 89] and a set of FD-configuring syscalls that demonstrates their feasibility. As this is not an exhaustive list of syscalls that an attacker can exploit, EINSTEIN is easily extendable to handle more syscalls, and to target different classes of syscalls [50, 63, 92].

Type of syscall	Chaining type	Syscalls
Security-sensitive syscalls	Does not chain	`execve, execveat, mprotect, mremap, remap_file_pages`
	Input: File FD	`mmap`
	Input: File or socket FD	`pwrite64, pwritev, pwritev2, sendfile, write, writev`
	Input: Socket FD	`sendmmsg, sendmsg, sendto`
FD-configuring syscalls	Output: File FD	`creat, open, openat, openat2`
	Output: File or socket FD	`dup, dup2, dup3`
	Output: Socket FD	`bind, connect, setsockopt, socket, socketpair`

5.3 Gadget Analyzer

To build exploits for the identified gadget chains, one could employ a heavyweight analysis that determines the exact constraints of all attacker dataflows e.g., via symbolic execution. However, as EINSTEIN aims to identify low-effort attacks, our gadget analyzer takes a lightweight approach that targets *identity dataflows*, i.e., dataflows where the value at the sink is identical to the value at the source. As we show in Section 7.1, this already yields a rich attack surface.

In our case, a *sink value* is a tainted syscall argument, and a *source value* is the data it originates from in the memory snapshot. We can map *sink values* to their corresponding *source values* by checking the *sink value*'s taint color, which contains its *source address* in the snapshot. In other words, we use the taint color of each reported syscall argument to identify the value it originates from in the memory snapshot. If these values are equivalent, then the argument has an identity flow.

Then, to exploit the identity flows, we generate $(addr, val)$ pairs to overwrite some *source value*. The specific exploit pairs that we may generate depend on a particular attacker's goals and the particular target program. Hence, for simplicity, we generate generic per-syscall exploits, e.g., with `execve` creating a particular file, or `write` writing a particular string. If the user wishes, the exploits may be modified from our exploitation tooling.

5.4 Exploitation Tooling

Because our gadget analyzer forgoes complex constraint solving, we consider the set of $(addr, val)$ pairs that it generates to only make up a *candidate* exploit. In order to confirm it as a *working* exploit, we must confirm that our exploit indeed stems from an identify data flow not violating any program invariants. Hence, rather than e.g., symbolically verifying all possible constraints that our exploit may effect, we punt the problem to the program itself—i.e., to a real, concrete execution. In particular, our exploitation tooling simply re-runs the target program and confirms whether the exploit is indeed successful. It does this by: (1) rewriting the data specified by the $(addr, val)$ pairs from the arbitrary memory write primitive; then (2) sending some workload to the target program to trigger the gadget chain; and finally (3) checking whether the exploit achieved its desired outcome. If the exploit is successful, then we confirm it to indeed be a *working* exploit.

We observe that some candidate exploits have a better chance of being successfully confirmed than others. Fortunately, because our approach casts such a wide net—e.g., by tracking *any possible* attacker data to *any combination* of security-sensitive syscalls—we have plenty of candidate exploits at our disposal. We prioritize those gadgets that we consider to be more robust and stronger attack primitives, e.g., gadgets that (1) exploit higher-value syscalls (e.g., `execve`), (2) exploit multiple syscall arguments, (3) overwrite deterministic addresses (e.g., global data), and (4) exploit syscall arguments with larger identity flows (for our experiments, we target identity flows that span at least 4 bytes).

6 Implementation

We base the implementation of our DTA engine on libdft [49]. Like other analyses that use libdft, the runtime component of EINSTEIN (which specifies the taint policies) is statically linked with our libdft variant. We invoke the target program with our tool enabled and with ASLR disabled. Although our threat model already assumes an attacker that bypasses ASLR, disabling ASLR for the analysis does provide a couple of implementation-level benefits: (1) it allows us to enforce 32-bit addressing, so that we can compress our tags from 64 bits to 32 bits; and (2) it causes deterministic addresses to stay constant across multiple runs, which, while not important for an attacker armed with an information leak, it does simplify the confirmation of candidate exploits that target such addresses. We expand upon other parts of our implementation in Appendix A.

7 Evaluation

We evaluate EINSTEIN in terms of attack surface and performance on an AMD Ryzen 9 3950X CPU with 128GB of RAM running Ubuntu 22.04.3 LTS (kernel v6.2). To run Newton [90], we use an Intel Xeon Silver 4108 CPU with 32GB of RAM running Ubuntu 16.04.7 LTS (kernel v4.3).

Table 3: Number of gadgets found per syscall and argument type.

Syscall*	Total covered	Total covered where this argument has a dataflow from attacker data and the percentage of those that are identity dataflows.					
		Arg. 1	Arg. 2	Arg. 3	Arg. 4	Arg. 5	Arg. 6
`execve`	11	7 (86%)	7 (86%)	7 (86%)			
`mmap`	787	55 (5%)	441 (8%)	55	16 (100%)	17	73
`mprotect`	144	97 (68%)	98 (27%)	1			
`mremap`	11	11	7	11	-	-	
`pwrite64`	1270	1217 (3%)	741 (47%)	399 (19%)	506 (36%)		
`pwritev`	10	10 (100%)	10 (10%)	-	10 (20%)		
`sendfile`	1	-	-	1	1		
`sendmmsg`	2	2	2	-	-		
`sendmsg`	12	5 (100%)	3 (100%)	-			
`sendto`	431	389 (7%)	410 (38%)	408 (6%)	-	-	-
`write`	3083	768 (74%)	2877 (91%)	1265 (10%)			
`writev`	754	244 (18%)	742 (76%)	-			

* Did not cover `execveat`, `pwritev2`, or `remap_file_pages`.

Table 4: Number of gadgets found and performance per target program.

Target program	*Non-control data* gadgets (% identity dataflow)	*Control data* gadgets* (% identity dataflow)		Runtime (h:mm:ss)		Peak PSS (MB)			Code coverage
		With register operand	With memory operand	Baseline	EINSTEIN	Baseline	EINSTEIN (1 color per tagset)	EINSTEIN (16 colors per tagset)	
`httpd`	1834 (97%)	2082 (42%)	15,179 (94%)	0:04:00	0:35:08	26	1560	3129	27.3%
`lighttpd`	92 (98%)	50 (34%)	155 (94%)	0:00:05	0:01:51	3	176	257	27.8%
`nginx`	1623 (82%)	503 (60%)	564 (99%)	0:08:41	3:27:05	24	479	848	49.1%
`postgres`	2105 (27%)	1382 (11%)	4690 (13%)	0:00:56	1:47:27	175	1568	11,621	46.5%
`redis`	218 (84%)	160 (22%)	53 (100%)	0:04:01	1:01:13	191	3257	32,188	33.6%

* Mitigated by code reuse defenses.

Programs and drivers. We target the web servers `httpd`, `lighttpd`, and `nginx`; and database servers `postgres` and `redis`—all of which have been shown to be at risk of memory write vulnerabilities [1–6]. Although we follow previous work by targeting server applications [19, 54, 61, 62, 76, 89–91], we note that even programs with more limited user-input interaction and few obviously dangerous syscalls (like `execve`), may well be at risk. Because EINSTEIN's approach only builds on a standard write vulnerability, it can indeed generalize to such applications.

We use each server's test suite to drive the analysis, since test suites are designed to test a variety of features and therefore cover a diverse set of code paths. If EINSTEIN is able to craft exploits based on simple test suites, it validates our claim that our approach greatly simplifies data-only attacks. Moreover, because `nginx` is a common target for exploitation case studies [57, 90, 94], we use our exploitation tooling to confirm the candidate exploits that EINSTEIN generates for `nginx`.

Program phase determination. As explained in Section 4, the phases of program execution (i.e., the initialization, quiescent, and processing phases) are application-dependent. Although more sophisticated approaches can determine the different phases of execution for a program (e.g., syscall monitoring [37]), all of our target servers initialize in a couple of seconds. Thus, we simply conservatively wait 10 seconds before establishing post-initialization quiescence.

7.1 Attack Surface Analysis

We investigate the attack surface uncovered by our approach based on: how effectively an attacker can control security-sensitive syscalls, and what an attacker can do with those syscalls. In Appendix B, we also evaluate how this attack surface compares with the attacks surface of control-flow hijacking attacks.

As in previous work [56], we present the number of syscalls in terms of *unique backtraces* since other possible metrics are problematic: presenting the *total executed syscalls* would artificially inflate the numbers (because the longer a program

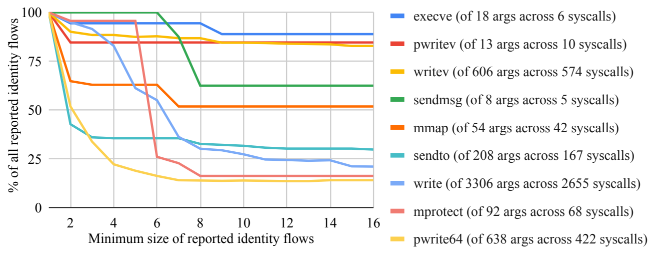


Figure 4: Percentage of all identity flows reported versus the minimum size of reported identity flows.

runs, the more syscalls it executes); while presenting *unique callsites* would artificially deflate the numbers (because a single syscall site may be called from many parts of a program).

Controllability. To evaluate the degree to which an attacker can control syscall arguments, we first refer to Table 3, which presents the number of tainted syscalls arguments, and in particular, the percentage of those with an identity flow. The first observation we make is that *most arguments have an identity flow*. Unsurprisingly, many of the identity flows are for data types that are generally treated as "immutable", e.g., the strings used in: the `pathname` and `argv` arguments to `execve` (86%); and the `buf` argument to `write` (91%) and `writev` (76%). Similarly, FD arguments tend to have a high rate of identity flow (e.g., 100% for `pwritev`, 100% for `sendmsg`, and 74% for `write`); this is also unsurprisingly because these FDs are often configured by syscalls that also take string-based arguments, e.g., the `pathname` to `open` and `creat`, and the `addr` to `connect` (which may be a string depending on its `sa_family`).

The second observation we make is based on Figure 4, which presents the lengths of the identity flows, broken down by syscall. We observe that many of the identity flows span at least 16 bytes, and hence, much of the data that an attacker can corrupt *remains unchanged* all the way to the vulnerable syscall. Given the prevalence of attacker data remaining unchanged to so many syscall arguments, we conclude that our data-only attacks are *low-effort*.

Syscall-based exploitation. To evaluate what an attacker can do with these controllable syscalls, we once again refer to Table 3 and observe that *every argument can be attacker-tainted*. Even for syscall arguments that we do not expect to be attacker-tainted (e.g., `mprotect`'s `prot` and `sendmsg`'s `flags` arguments, which are typically compiled down into a constant), we still encounter cases of tainted arguments. For example, we identify a case where `pthread_create` calls `mprotect` with a `prot` argument that has an identity flow from attacker-controllable data on the heap.

Next, we refer to Table 5, which presents our confirmed exploits for `nginx`, and observe that they offer *many primitives to an attacker*. For example, a vulnerable `execve` gives us a CODE-EXECUTION primitive; vulnerable file-configuring syscalls (e.g., `openat`) combined with vulnerable file-write syscalls (e.g., `write`) give us 17 WRITE-WHAT-WHERE primitives; vulnerable socket-configuring syscalls (e.g., `connect`) combined with vulnerable socket-write syscalls (e.g., `sendmsg`) give us 41 SEND-WHAT-WHERE primitives; and vulnerable syscalls combined with non-vulnerable syscalls give us less powerful, but still more numerous primitives (i.e., 885 total WRITE-WHAT, WRITE-WHERE, SEND-WHAT, and SEND-WHERE primitives). In comparison, FlowStitch [41]—a previous approach that also (i) generates syscall-based exploits, and (ii) assumes an arbitrary memory write primitive in its evaluation of `nginx` [2]—is hampered by concolic execution's poor coverage, and as a result, only generates 2 exploits for `nginx`. Because EINSTEIN identifies such a broad range of tainted arguments, and generates a variety attack primitives, we conclude that our data-only attacks are *diverse*.

Table 5: Confirmed exploits for `nginx`.

Attack primitive	Count
CODE-EXECUTION	1
WRITE-WHAT-WHERE	17
WRITE-WHAT	375
WRITE-WHERE	79
SEND-WHAT-WHERE	41
SEND-WHAT	372
SEND-WHERE	59
Total	944

7.2 Performance Analysis

We investigate the performance of EINSTEIN along two dimensions. First, we investigate the overall performance of EINSTEIN per target program, as shown in Table 4. The baseline that we compare to is the vanilla target program, i.e., without any instrumentation.

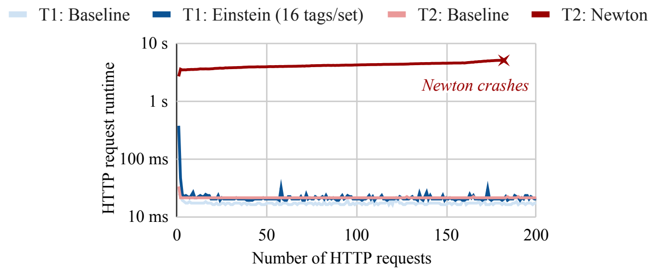
Second, we investigate the performance of EINSTEIN's DTA engine. To do this, we compare the overhead of EINSTEIN to the overhead of Newton [90], which, in contrast to EINSTEIN, does not have the DTA engine optimizations described in Section 5.1. Specifically, its DTA engine: (1) accesses its tagmap via a *page table walk*, (2) initializes its tagmap via a *software-based MMU check*, and (3) represents tags as *unbounded sets*. Figure 5 presents the overheads of Newton, EINSTEIN, and their baselines from running `nginx` under a minimal configuration and processing multiple HTTP requests. We note two issues that led to this evaluation setup: (1) any moderately complex workload (e.g., running with a configuration that loads more than a few modules, or processing an HTTPS request) caused Newton to immediately crash, so we only run with a minimal configuration and process simple requests; and (2) for a variety of technical reasons (e.g., Newton requiring an older kernel version, full 32-bit support, etc.) we could not evaluate EINSTEIN and Newton on the same machine, so we present the baseline overheads on both machines; T1 is EINSTEIN's machine, and T2 is Newton's machine.

Runtime overhead. As we see in Table 4, the runtime overhead of EINSTEIN ranges from 8x–26x. Our extensive experiments, including full test suites, confirm that the DTA engine easily scales to a complete testing campaign. Moreover, we see in Figure 5a that EINSTEIN handles `GET` requests at least two orders of magnitude faster than Newton. We attribute EINSTEIN's speedup primarily to the way it accesses the tagmap: on every access, it performs a fast shadow memory lookup, whereas Newton navigates its page table structure. Because our DTA engine optimizations enables a practical analysis platform, we consider the runtime overhead *acceptable*.

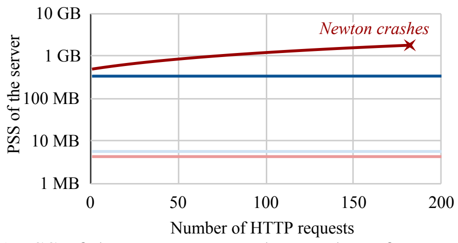
Memory overhead. Next, we evaluate the memory overhead of EINSTEIN when using both 1 color per tagset and when using 16 colors per tagset (i.e., the default). With 1 color/tagset, tags are easily prone to overfill, because adding any more than 1 color to a tagset results in it no longer tracking individual colors. Hence, this configuration cannot build candidate exploits for many gadget chains, because many syscall arguments' tags are overfilled. Nonetheless, it still identifies all the candidate gadget chains that the 16 colors/tagset configuration identifies.

We find in Table 4 that the memory overhead of EINSTEIN with 1 color/tagset is 9–60x relative the uninstrumented baseline. This overhead accounts for the internal data structures of EINSTEIN, `libdft64-ng`, etc.—most significant of which is the tagmap, which contains a tagset for every byte of program memory. If we increase the number of colors/tagset from 1 to 16, we would expect the resulting memory overhead to be 16x in the worst case. However, we find that our target databases have a 7.4–9.9x overhead, and our target web servers only have a 1.6–2.0x overhead. We attribute worse overheads to target applications having more spatially fine-grained access patterns, because they need to page in more of the tagmap per page of program memory.

Next, we see in Figure 5b that EINSTEIN has a constant memory overhead, whereas Newton's memory overhead grows for each additional `GET` request, until it eventually crashes from reaching the 2 GB limit for 32-bit processes. We attribute this difference to the way the DTA engines represent tags: whereas EINSTEIN uses *bounded* sets of 16 tags/set, Newton uses *unbounded* sets. As a result, Newton will combine any new colors with a tag until it runs out of memory. In principle, this has a worst case $O(n^2)$ memory overhead (where n is the total program memory), if every byte of program data is combined with every other byte of program data. Hence, an application such as `nginx`, which requires about 16 MB to run normally, would in the worst case, require 256 TB to run with Newton. In reality, however, the 2 GB limit masks this overhead, and it simply crashes well before this limit is reached. We conclude that because of our DTA engine optimizations and because of its configurable number of colors per tagset (i.e., to accommodate the tradeoff between total available memory and number of exploits generated), our memory overhead is *acceptable*.



(a) Runtime of a request versus the number of requests.



(b) PSS of the server versus the number of requests.

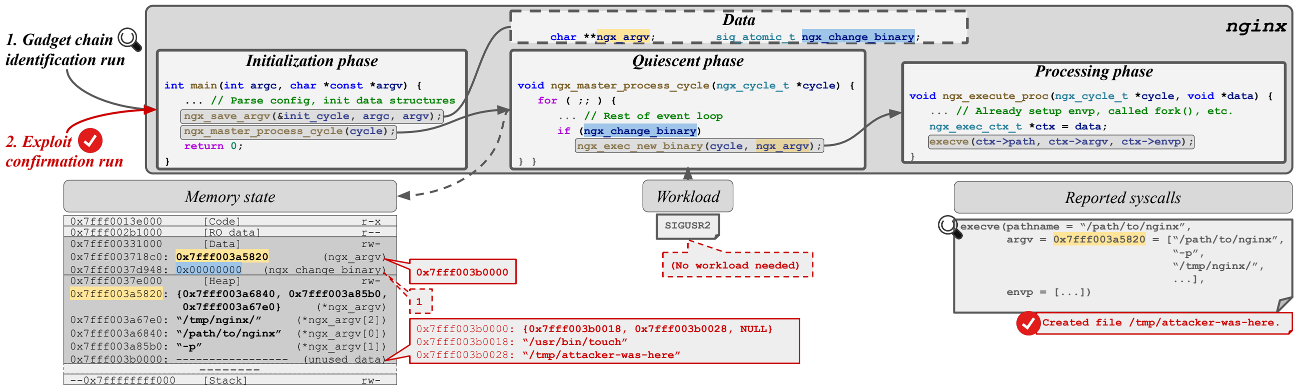
Figure 5: Comparison of running `nginx` with EINSTEIN and Newton [90] to running `nginx` as is (i.e., the baseline).

Coverage. As with any dynamic analysis, its efficacy is dependent on program coverage: poor coverage will generally yield poor results, and vice-versa. Yet, even though the test suites only covered 27–49% of the programs' code, EINSTEIN still uncovered many gadgets. Future work to increase code coverage, e.g., by exploiting syscall-guard variables [95], would undoubtedly *yield more gadgets*.

8 Case Studies

We have demonstrated that EINSTEIN builds diverse, low-effort attacks for popular servers. To demonstrate that such attacks pose a serious threat, we present two case studies of attacks against `nginx`. For each case study, we will: (1) explain how EINSTEIN identifies the gadget chain, builds a candidate exploit for it, and confirms the exploit's effectiveness; and (2) discuss how the exploit bypasses state-of-the-art defenses.

Defenses targeted. In considering which defenses to center our discussion around, we recall our explanation in Section 2 about the *practical* and *comprehensive* defenses against vulnerable *defs* and *uses*. Because we assume the existence of a memory write vulnerability, any *def*-centric defenses (e.g., memory safety, DSR, and memory error scanning) are implicitly out of scope. Moreover, because the *comprehensive* defenses (e.g., memory safety, full DSR, and full DFI) are not widely deployed, we also consider them out of scope. Hence, we center our discussion around the *practical use*-centric defenses, i.e., syscall filtering [36, 37, 46, 69] and selective



```
char **ngx_argv;            sig_atomic_t ngx_change_binary;
```

```
int main(int argc, char *const *argv) {
    ... // Parse config, init data structures
    ngx_save_argv(&init_cycle, argc, argv);
    ngx_master_process_cycle(cycle);
    return 0;
}
```

```
void ngx_master_process_cycle(ngx_cycle_t *cycle) {
  for ( ;; ) {
    ... // Rest of event loop
    if (ngx_change_binary)
      ngx_exec_new_binary(cycle, ngx_argv);
} }
```

```
void ngx_execute_proc(ngx_cycle_t *cycle, void *data) {
  ... // Already setup envp, called fork(), etc.
  ngx_exec_ctx_t *ctx = data;
  execve(ctx->path, ctx->argv, ctx->envp);
}
```

```
0x7fff0013e000        [Code]                    r-x
0x7fff002b1000        [RO data]                 r--
0x7fff00331000        [Data]                    rw-
0x7fff003718c0: 0x7fff003a5820          (ngx_argv)
0x7fff0037d948: 0x00000000     (ngx_change_binary)
0x7fff0037e000        [Heap]                    rw-
0x7fff003a5820: {0x7fff003a6840, 0x7fff003a85b0,
                 0x7fff003a67e0}       (*ngx_argv)
0x7fff003a67e0: "/tmp/nginx/"       (*ngx_argv[2])
0x7fff003a6840: "/path/to/nginx"    (*ngx_argv[0])
0x7fff003a85b0: "-p"                (*ngx_argv[1])
0x7fff003b0000: -----------------    (unused data)
                   --------
--0x7ffffffff000      [Stack]                   rw-
```

```
0x7fff003b0000
```

```
1
```

```
0x7fff003b0000: {0x7fff003b0018, 0x7fff003b0028, NULL}
0x7fff003b0018: "/usr/bin/touch"
0x7fff003b0028: "/tmp/attacker-was-here"
```

```
execve(pathname = "/path/to/nginx",
       argv = 0x7fff003a5820 = ["/path/to/nginx",
                                "-p",
                                "/tmp/nginx/",
                                ...],
       envp = [...])
```

Figure 6: An `execve`-based gadget in `nginx` that EINSTEIN identified and built an attack for.

DFI [46, 81]. As we demonstrate that such *practical* defenses are fundamentally unsecure, we conclude that more *comprehensive* defenses should be deployed.[4]

8.1 Bypassing Syscall Filtering

This case study exploits an `execve` gadget in `nginx`'s live binary upgrade feature, wherein `nginx` replaces its own running binary while maintaining all of its connections. Figure 6 walks through how we identify and exploit the gadget with EINSTEIN.

Identification and exploitation. First, when starting up, `nginx` saves its `argv` to the global variable `ngx_argv`, which is subsequently tainted by EINSTEIN. Next, when sending workloads to `nginx`, EINSTEIN sends the `SIGUSR2` signal to the `nginx` master process, which initiates the live binary upgrade. This raises the `ngx_change_binary` condition, which causes `nginx` to call `ngx_exec_new_binary` with `ngx_argv` as an argument. Eventually, this results in `nginx` calling `execve`, passing to it the same `argv` that it had initially saved during start up. EINSTEIN identifies the attacker-tainted argument to `execve` and reports it.

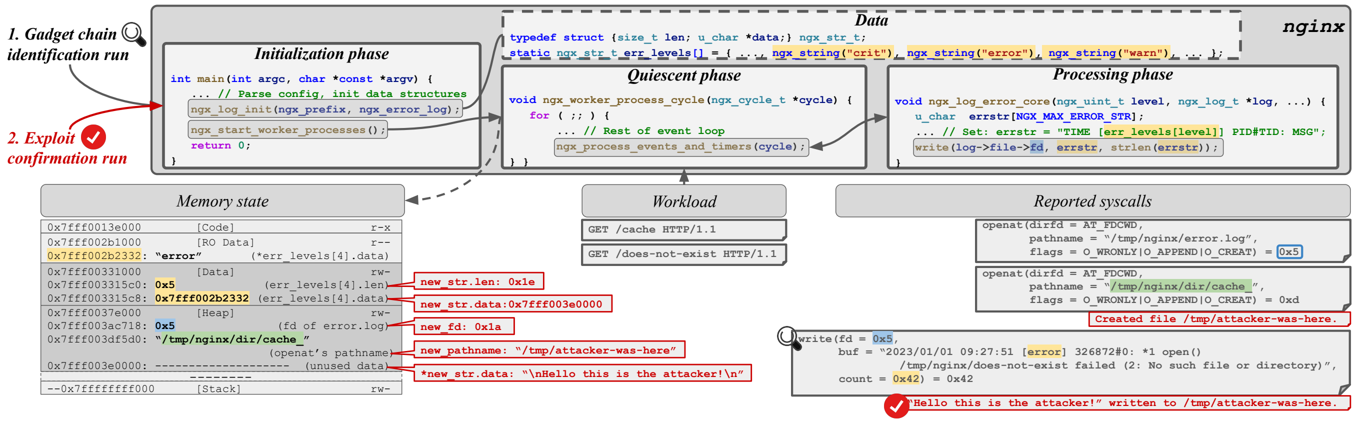
EINSTEIN builds an exploit for this by first identifying the identity dataflow from `execve`'s `argv` argument to the `ngx_argv` global variable. Then, EINSTEIN builds an "arbitrary code execution" candidate exploit by generating $(addr, val)$ pairs that overwrite `ngx_argv` to point to an array of two strings: `"/usr/bin/touch"`, `"/tmp/attacker-was-here"`, and a `NULL` terminator. After re-running `nginx` to confirm this exploit, it indeed creates the file `/tmp/attacker-was-here`, indicating that our exploit was successful.

Because this gadget is triggered server-side (via `SIGUSR2`) rather than client-side (via a network request), it would appear at first glance that a remote attacker cannot trigger it—rather, the attacker would have to wait until the `nginx` process upgrades itself. However, our exploitation tooling (which parses the gadget's backtrace) makes it immediately clear to us that we can indeed trigger this gadget remotely. In particular, by trivially overwriting the tainted global variable `ngx_change_binary`, we can force `nginx` to take the conditional branch before `ngx_exec_new_binary`, and therefore call `execve`. Hence, with just a bit of manual inspection, an attacker can escalate non-triggerable gadgets such as this into triggerable gadgets. We note that this approach to "legal" control-flow hijacking differs from previous data-only approaches for a couple reasons. First, it is *optional*, because this gadget can also be triggered: (1) by `nginx`, when it upgrades itself, and (2) by the attacker, by hijacking a tainted `kill` syscall. Second, it is considerably *lower effort*, because we simply flip a single branch that precedes our already-identified gadget, whereas other approaches analyze all possible branches in an attempt to identify new gadgets [43, 45, 75].

Discussion. The objective of syscall filter-based defenses [36, 37, 69] is to disable high-value syscalls, e.g., `execve`. However, *syscalls filters are fundamentally imprecise* because they cannot disable legitimate syscalls. In other words, if `nginx` can legitimately invoke `execve`, then a defense cannot disable `execve`—otherwise it would break `nginx`. EINSTEIN exploits this imprecision because it does not divert control flow, and hence, it only covers legitimate syscalls, thereby eliding syscall filter-based defenses.

In principle, if a security-conscious sysadmin designed a configuration for `nginx` that completely disabled all legitimate uses of `execve` (e.g., by disabling CGI, live binary upgrade, etc.), then a syscall filter could indeed disable `execve`, thereby mitigating this gadget. However, in practice, this is nearly impossible because: (1) it is not possible to disable some features from a configuration (e.g., live binary upgrade); and (2) it is non-trivial to determine that no possible program point could invoke `execve` (e.g., due to the presence of dynamically loaded libraries and modules). This observation is confirmed by the evaluation of state-of-the-art syscall filters on `nginx`: none of them are able to completely disable `execve` [36, 37, 69].

[4] Indeed, developers may deploy the comprehensive (but costly) defenses to thwart EINSTEIN's attacks. Moreover, they may use EINSTEIN's reports to harden their program (e.g., by sanitizing particular syscall arguments).



```
typedef struct {size_t len; u_char *data;} ngx_str_t;
static ngx_str_t err_levels[] = { ..., ngx_string("crit"), ngx_string("error"), ngx_string("warn"), ... };
```

```
int main(int argc, char *const *argv) {
  ... // Parse config, init data structures
  ngx_log_init(ngx_prefix, ngx_error_log);
  ngx_start_worker_processes();
  return 0;
}
```

```
void ngx_worker_process_cycle(ngx_cycle_t *cycle) {
  for ( ;; ) {
    ... // Rest of event loop
    ngx_process_events_and_timers(cycle);
} }
```

```
void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ...) {
  u_char  errstr[NGX_MAX_ERROR_STR];
  ... // Set: errstr = "TIME [err_levels[level]] PID#TID: MSG";
  write(log->file->fd, errstr, strlen(errstr));
}
```

```
0x7fff0013e000          [Code]                     r-x
0x7fff002b1000          [RO Data]                  r--
0x7fff002b2332: "error"         (*err_levels[4].data)
0x7fff00331000          [Data]                     rw-
0x7fff003315c0: 0x5               (err_levels[4].len)   new_str.len: 0x1e
0x7fff003315c8: 0x7fff002b2332  (err_levels[4].data)   new_str.data:0x7fff003e0000
0x7fff0037e000          [Heap]                     rw-
0x7fff003ac718: 0x5                 (fd of error.log)   new_fd: 0x1a
0x7fff003df5d0: "/tmp/nginx/dir/cache_"
                                  (openat's pathname)   new_pathname: "/tmp/attacker-was-here"
0x7fff003e0000: --------------------   (unused data)   *new_str.data: "\nHello this is the attacker!\n"
                         --------
--0x7ffffffff000        [Stack]                    rw-
```

```
GET /cache HTTP/1.1
GET /does-not-exist HTTP/1.1
```

```
openat(dirfd = AT_FDCWD,
       pathname = "/tmp/nginx/error.log",
       flags = O_WRONLY|O_APPEND|O_CREAT) = 0x5
openat(dirfd = AT_FDCWD,
       pathname = "/tmp/nginx/dir/cache_",
       flags = O_WRONLY|O_APPEND|O_CREAT) = 0xd
                              Created file /tmp/attacker-was-here.
write(fd = 0x5,
      buf = "2023/01/01 09:27:51 [error] 326872#0: *1 open()
               /tmp/nginx/does-not-exist failed (2: No such file or directory)",
      count = 0x42) = 0x42
           "Hello this is the attacker!" written to /tmp/attacker-was-here.
```

Figure 7: An openat-to-write-based gadget chain in nginx that EINSTEIN identified and built an attack for.

8.2 Bypassing Selective DFI

This case study exploits an openat to write gadget chain in nginx's error log. Figure 7 walks through how we identify and exploit the gadget with EINSTEIN.

Identification and exploitation. After nginx enters its quiescent phase, EINSTEIN taints three nginx objects in particular: (1) the err_levels[] global array, which contains pointers to logging strings (e.g., "crit", "error", "warn") and their lengths; (2) the pathname of a configuration-defined cache file, which nginx opens upon requests for /cache; and (3) the FD of nginx's error log. Then, EINSTEIN sends requests to nginx, two of which trigger the syscalls of our gadget chain. First, it sends a GET /cache request, which causes nginx to open the cache file. It opens it via an openat with a tainted pathname. EINSTEIN then adds the created file to its directly-controllable FD table. Second, EINSTEIN sends a GET /does-not-exist request, which causes nginx to write an "error" message to the error log. It does this via a write with a tainted fd, buf, and count. Because the FD argument is tainted, and there is a file in the directly-controllable FD table, EINSTEIN determines that the write could be redirected to another file, thereby identifying the chain from the openat to the write.

EINSTEIN builds an exploit for this by first identifying an identity flow between: (1) openat's pathname and the cache pathname on the heap, (2) write's fd and the error log's FD on the heap, and (3) write's buf and the pointer to the "error" string in global data. Then, EINSTEIN builds a "write-what-where on the filesystem" candidate exploit by generating $(addr, val)$ pairs that overwrite: (1) the cache file's pathname to a file "/tmp/attacker-was-here", (2) the error log's FD to the cache file's FD, and (3) the pointer to the "error" string to a pointer to a "Hello this is the attacker" string. Next, we re-run EINSTEIN to confirm this exploit, but unfortunately, it only writes "Hello" to our file. Upon closer inspection as to why, we quickly notice from the tagset of write's count argument that it depends on the "error" string's length—i.e., 0x5, thereby explaining why only the first five characters of our target string were written to the file. Our analyzer did not initially target the string length because it does not have an identity dataflow to write's count, since their values differ. Hence, we modify our exploit to also overwrite the err_levels[] string length to 0x1e, i.e., the length of our string. As a result, we are able successfully write our entire attacker-specified string to our attacker-specified file.

Discussion. The objective of selective DFI is to only enforce DFI for select *def-use* chains, e.g., for syscall argument *uses* [46]. However, *selective DFI is fundamentally imprecise* because: (1) it leaves many dataflows unsecure (i.e., it is "selective"), and (2) for any complex dataflows it does attempt to secure, it must statically over-approximate the set of legal *def*s for a given *use*, or otherwise risk breaking the target program. EINSTEIN exploits this imprecision because it uses a lightweight dynamic analysis, which encounters no issues with complex dataflows.

Specifically, if the selective DFI's points-to analysis cannot determine all the possible *def*s for some loaded data—e.g., due to notoriously difficult interprocedural analyses, pointer aliasing problems, etc. [40]—then it must assume that *any def* is legal. For example, in our gadget, the dataflow from the *def* of the "error" string to the *use* of write's buf is difficult to resolve: err_levels[] has four address-taken locations that may be passed interprocedurally up to five calls deep (e.g., to a custom implementation of the variadic sprintf). Hence, securing every possible *def* for write's buf is non-trivial, and over-approximation would likely incur a significant performance overhead. Unsurprisingly, recent work that applies selective DFI to syscall arguments does not attempt to secure writes [46]. Finally, we note that this gadget is also resistant to syscall filtering because numerous legitimate program features rely on openat and write.

9 Limitations

Coverage. The coverage of our analysis is limited for a few reasons. First, like any dynamic analysis, our code coverage is not complete. Nonetheless, we still find many exploits, and as Section 7.2 explains, further work into improving code coverage would also identify more attacks. Second, like other DTA engines, we do not track implicit flows. However, we consider this acceptable, because implicit flows are less useful to an attacker since they typically only propagate one bit of data. Third, EINSTEIN does not build every kind of data-only attack, e.g., those that aim for Turing-completeness. Instead, EINSTEIN aims for a simpler, but more powerful primitive: controllable syscalls. Identifying and eliminating these will raise the bar for attackers significantly.

Constraints. Like other approaches that use DTA, our analysis is limited as it does not track all possible dataflow constraints in two ways. First, it does not track exactly how arbitrary program operations (e.g., arithmetic) affects attacker data. However, this is not an issue for us, because our identity dataflow analysis filters out these more complex gadgets, and we are still left with plenty of candidate gadgets. Second, it does not track exactly how attacker data affects the program's path. However, this is also not an issue, because our exploit confirmation filters out these more complex candidate exploits, and we are still left with plenty of working exploits.

10 Conclusion

We presented EINSTEIN, a lightweight approach to building practical attacks that are out of reach of prior solutions. We use EINSTEIN to build hundreds of data-only attacks and present two low-effort case studies that bypass state-of-the-art defenses. We conclude that data-only attacks are well within reach of attackers, and hence, vendors should consider stronger defenses such as full DFI and memory safety.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. We would also like to thank Alyssa Milburn for helping with the initial shadow memory implementation and Mans van Someren for fixing a number of libdft bugs. This work was supported by the Dutch Research Council (NWO) through project "INTERSECT" and "Theseus".

References

[1] "CVE-2007-4727," https://nvd.nist.gov/vuln/detail/CVE-2007-4727.

[2] "CVE-2013-2028," https://nvd.nist.gov/vuln/detail/CVE-2013-2028.

[3] "CVE-2021-32027," https://nvd.nist.gov/vuln/detail/CVE-2021-32027.

[4] "CVE-2022-23943," https://nvd.nist.gov/vuln/detail/CVE-2022-23943.

[5] "CVE-2022-41741," https://nvd.nist.gov/vuln/detail/CVE-2022-41741.

[6] "CVE-2023-36824," https://nvd.nist.gov/vuln/detail/CVE-2023-36824.

[7] "DataFlowSanitizer (Clang 12 documentation)," https://releases.llvm.org/12.0.0/tools/clang/docs/DataFlowSanitizerDesign.html.

[8] "DataFlowSanitizer (Clang 13 documentation)," https://releases.llvm.org/13.0.0/tools/clang/docs/DataFlowSanitizerDesign.html.

[9] "KernelDataFlowSanitizer," https://github.com/vusec/kdfsan-linux/.

[10] "KernelMemorySanitizer," https://github.com/google/kmsan.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications," *TISSEC*, vol. 13, no. 1, 2009.

[12] S. Ahmed, H. Liljestrand, H. Jamjoom, M. Hicks, N. Asokan, and D. D. Yao, "Not All Data are Created Equal: Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks," in *USENIX Security*, 2023.

[13] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *S&P*, 2008.

[14] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors," in *USENIX Security*, 2009.

[15] S. Andersen and V. Abella, "Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention," http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[16] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek *et al.*, "Hardware Assisted Randomization of Data," in *RAID*, 2018.

[17] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, "Deterministic Process Groups in dOS," in *OSDI*, 2010.

[18] S. Bhatkar and R. Sekar, "Data Space Randomization," in *DIMVA*, 2008.

[19] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *S&P*, 2014.

[20] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A New Class of Code-Reuse Attack," in *AsiaCCS*, 2011.

[21] E. Bosman and H. Bos, “Framing Signals——A Return to Portable Shellcode,” in *S&P*, 2014.

[22] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, “Data Randomization,” Technical Report TR-2008-120, Microsoft Research., Tech. Rep., 2008.

[23] S. A. Carr and M. Payer, “DataShield: Configurable Data Confidentiality and Integrity,” in *AsiaCCS*, 2017.

[24] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *OSDI*, 2006.

[25] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-Oriented Programming without Returns,” in *CCS*, 2010.

[26] P. Chen and H. Chen, “Angora: Efficient Fuzzing by Principled Search,” in *S&P*, 2018.

[27] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-Control-Data Attacks Are Realistic Threats,” in *USENIX Security*, 2005.

[28] L. Cheng, S. Ahmed, H. Liljestrand, T. Nyman, H. Cai, T. Jaeger, N. Asokan, and D. Yao, “Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches,” *TOPS*, vol. 24, no. 4, 2021.

[29] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. Yao, “Exploitation Techniques and Defenses for Data-Oriented Attacks,” in *SecDev*, 2019.

[30] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, “ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks,” in *NDSS*, 2014.

[31] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “PointGuardTM: Protecting Pointers From Buffer Overflow Vulnerabilities,” in *USENIX Security*, 2003.

[32] T. H. Dang, P. Maniatis, and D. Wagner, “The Performance Cost of Shadow Stacks and Stack Canaries,” in *AsiaCCS*, 2015.

[33] J. Davis, A. Thekumparampil, and D. Lee, “Node.fz: Fuzzing the Server-Side Event-Driven Architecture,” in *EuroSys*, 2017.

[34] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, “Efficient Protection of Path-Sensitive Control Security,” in *USENIX Security*, 2017.

[35] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR,” in *MICRO*, 2016.

[36] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, “Temporal System Call Specialization for Attack Surface Reduction,” in *USENIX Security*, 2020.

[37] S. Ghavamnia, T. Palit, and M. Polychronakis, “C2C: Fine-grained Configuration-driven System Call Filtering,” in *CCS*, 2022.

[38] F. Gorter, E. Barberis, R. Isemann, E. van der Kouwe, C. Giuffrida, and H. Bos, “FloatZone: Accelerating Memory Error Detection using the Floating Point Unit,” in *USENIX Security*, 2023.

[39] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, “ASLR on the Line: Practical Cache Attacks on the MMU,” in *NDSS*, 2017.

[40] M. Hind, “Pointer Analysis: Haven’t We Solved This Problem Yet?” in *PASTE*, 2001.

[41] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, “Automatic Generation of Data-Oriented Exploits,” in *USENIX Security*, 2015.

[42] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, “Enforcing Unique Code Target Property for Control-Flow Integrity,” in *CCS*, 2018.

[43] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks,” in *S&P*, 2016.

[44] R. Hund, C. Willems, and T. Holz, “Practical Timing Side Channel Attacks Against Kernel Space ASLR,” in *S&P*, 2013.

[45] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, “Block Oriented Programming: Automating Data-Only Attacks,” in *CCS*, 2018.

[46] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, and C. Min, “Protect the System Call, Protect (most of) the World with BASTION,” in *ASPLOS*, 2023.

[47] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, “Cyclone: a safe dialect of C,” in *ATC*, 2002.

[48] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, “Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel,” in *NDSS*, 2022.

[49] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, “libdft: Practical Dynamic Data Flow Tracking for Commodity Systems,” in *VEE*, 2012.

[50] K. Koning, H. Bos, and C. Giuffrida, "Secure and Efficient Multi-variant Execution Using Hardware-assisted Process Virtualization," in *DSN*, 2016.

[51] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng, "Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses," in *TrustCom*, 2015.

[52] C. Lattner and V. S. Adve, "Transparent Pointer Compression for Linked Data Structures," in *MSP*, 2005.

[53] T. Liu, G. Shi, L. Chen, F. Zhang, Y. Yang, and J. Zhang, "TMDFI: Tagged Memory Assisted for Fine-grained Data-Flow Integrity towards Embedded Systems against Software Exploitation," in *TrustCom*, 2018.

[54] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks," in *CCS*, 2015.

[55] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.

[56] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *TSE*, vol. 47, no. 11, 2019.

[57] M. Morton, J. Werner, P. Kintis, K. Snow, M. Antonakakis, M. Polychronakis, and F. Monrose, "Security Risks in Asynchronous Web Servers: When Performance Optimizations Amplify the Impact of Data-Oriented Attacks," in *EuroS&P*, 2018.

[58] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *PLDI*, 2009.

[59] ——, "CETS: Compiler-Enforced Temporal Safety for C," in *ISMM*, 2010.

[60] J. Newsome and D. X. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *NDSS*, 2005.

[61] B. Niu and G. Tan, "Per-Input Control-Flow Integrity," in *CCS*, 2015.

[62] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking Holes in Information Hiding," in *USENIX Security*, 2016.

[63] OpenBSD, "pledge(2)," https://man.openbsd.org/pledge.2, 2016.

[64] T. Palit, F. Monrose, and M. Polychronakis, "Mitigating Data Leakage by Protecting Memory-resident Sensitive Data," in *ACSAC*, 2019.

[65] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis, "DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection," in *S&P*, 2021.

[66] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing," in *USENIX Security*, 2013.

[67] PaX Team, "PaX ASLR (Address Space Layout Randomization)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[68] J. Pewny, P. Koppe, and T. Holz, "Steroids for DOPed Applications: A Compiler for Automated Data-Oriented Programming," in *EuroS&P*, 2019.

[69] A. Quach, A. Prakash, and L. Yan, "Debloating Software through Piece-Wise Compilation and Loading," in *USENIX Security*, 2018.

[70] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz, "CoDaRR: Continuous Data Space Randomization against Data-Only Attacks," in *AsiaCCS*, 2020.

[71] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *NDSS*, 2017.

[72] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *ACSAC*, 2009.

[73] A. Sadeghi, S. Niksefat, and M. Rostamipour, "Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions," *JCVHT*, 2018.

[74] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *USENIX Security*, 2011.

[75] E. J. Schwartz, C. F. Cohen, J. S. Gennari, and S. M. Schwartz, "A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks," in *CCS*, 2020.

[76] J. Seibert, H. Okhravi, and E. Söderström, "Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code," in *CCS*, 2014.

[77] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *ATC*, 2012.

[78] J. Seward and N. Nethercote, "Using Valgrind to Detect Undefined Value Errors with Bit-Precision," in *ATC*, 2005.

[79] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *CCS*, 2007.

[80] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *CCS*, 2004.

[81] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," in *NDSS*, 2016.

[82] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-Assisted Data-flow Isolation," in *S&P*, 2016.

[83] E. H. Spafford, "The Internet Worm Program: An Analysis," *SIGCOMM CCR*, vol. 19, no. 1, 1989.

[84] M. Stamatogiannakis, P. Groth, and H. Bos, "Looking Inside the Black-Box: Capturing Data Provenance using Dynamic Instrumentation," in *IPAW*, 2015.

[85] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *CGO*, 2015.

[86] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," in *ASPLOS*, 2004.

[87] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *S&P*, 2013.

[88] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *USENIX Security*, 2014.

[89] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *CCS*, 2015.

[90] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later," in *CCS*, 2017.

[91] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level," in *S&P*, 2016.

[92] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz, "Secure and Efficient Application Monitoring and Replication," in *ATC*, 2016.

[93] D. Wagner and R. Dean, "Intrusion Detection via Static Analysis," in *S&P*, 2000.

[94] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and F. Geck, "PatchRNN: A Deep Learning-Based System for Security Patch Identification," in *MILCOM*, 2021.

[95] H. Ye, S. Liu, Z. Zhang, and H. Hu, "Viper: Spotting Syscall-Guard Variables for Data-Only Attacks," in *USENIX Security*, 2023.

[96] S. H. Yong and S. Horwitz, "Protecting C Programs from Attacks via Invalid Pointer Dereferences," in *ESEC*, 2003.

[97] W. D. Young and J. Mchugh, "Coding for a believable specification to implementation mapping," in *S&P*, 1987.

[98] M. Zhang and R. Sekar, "Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-World ROP Attacks," in *ACSAC*, 2015.

A Implementation Digression

In this appendix, we discuss the features we added and merged into our DTA engine to make it general-purpose and extendable. We base the implementation of our DTA engine on libdft [49], a DTA engine built on top of Intel Pin [55], which was released more than 10 years ago.

Merged features. Since libdft's release, many forks of it have emerged, each adding their own features to suite their own particular analyses [26, 71, 84, 90]. Unfortunately, no single version of libdft exists that supports all of these features, so we merge many of them into one *general-purpose* DTA engine, i.e., our libdft variant. In particular, we merge support for: (1) 64-bit instructions; (2) load pointer propagation, i.e., on a load instruction, propagating not only the tag of the loaded data to the output, but also the tag of the pointer; (3) tagging all data with its own address; (4) an interface to the process's memory map to e.g., check whether some memory segment is writable; and (5) a uniform logging interface, which supports applications that are multi-threaded and applications that redirect standard logging interfaces (e.g., `nginx`).

Added features. We also add several new features to libdft. The first set of features we add provide scalability (as discussed in Section 5.1). In particular, we add: (1) `userfaultfd`-based tagmap initialization, (2) bounded array-based tagsets, and (3) a shadow memory-based tagmap.

Figure 8 describes the layout of our shadow memory implementation. At a high level, the region from `MAIN_START` to `MAIN_END` is the main address space, and the region from `SHADOW_START` to `SHADOW_END` is the corresponding shadow address space. Because the region from `MAIN_START` to `MAIN_START+RESERVED_BYTES` would map into null shadow

```
==== 0x000000000000: SHADOW_START
...
==== 0x000000100000: SHADOW_START+RESERVED_BYTES
...
...
...
==== 0x............: SHADOW_END, MAIN_START
...
==== 0x............: MAIN_START+RESERVED_BYTES
...
...
==== 0x7fff00101000: BIN_START
...
==== 0x800000000000: MAIN_END
```

Figure 8: Shadow memory layout.

memory, we cannot use it for program memory. Hence, we reserve this region to store custom tags. For example, to taint a file read (as Angora does [26]), we can create unique tags in this reserved region for each byte of a file. Hence, by supporting custom, unique tags—e.g., for each byte of a file, network read, etc.—our shadow memory provides an *expressive* interface for other analyses.

The shadow memory layout assumes the target program is an `x86_64` binary built as position-independent (the default for most compilers), so that its stack and `mmap_base` are at the end of the address space. Next, the layout requires us to relink the target program such that its base address is at `BIN_START`. The value of `RESERVED_BYTES` and `SHADOW_END` depend on the tag size—i.e., the number of colors per tagset, and the size of each color (which depends on whether pointer tag compression is enabled).

The second set of features we add provide other various functionalities. First, we add support for recording memory snapshots. We do this by invoking `gcore` when tainting all memory to produce a core dump. Because the shadow memory comprises a significant portion of the address space, we advise the kernel to not dump shadow pages so that the core dumps are not too large. Moreover, to aid our exploitation tooling, we use `lldb` to also log all runtime symbol information when the snapshot is recorded. Second, we add an interface to receive miscellaneous runtime info. We use this interface to receive the most recently executed line of the program driver, so that we can include it in the syscall reports, thereby aiding our exploitation tooling in knowing the workload that covers a particular gadget chain. Third, we fix many instruction-level taint propagation rules, e.g., by: (1) considering the byte-level semantics of masking operations (e.g., `AND`, `OR`); (2) adding support for various vector instructions (e.g., `VZEROUPPER`, `PUNPCKLQDQ`); (3) correctly handling various x86-64 quirks (e.g., zero-extending a 32-bit `MOV`, but not an 8-, 16-, or 64-bit `MOV`); etc. Fourth, we update our libdft variant to use the newest version of Pin and to support newer syscalls (e.g., `execveat`, `openat2`).

B Control Data Attack Surface Comparison

In this appendix, we compare our *non-control data* attack surface to the *control data* attack surface. To do so, we re-implement the taint policies of Newton [90] (with our taint engine) for identifying control-flow hijacking attacks, and re-run our evaluation. We note that our non-control data gadgets are inherently more powerful than control data gadgets, because even if an attacker can divert a branch, the attacker still needs to divert it to some target code that e.g., invokes a syscall with controllable arguments. In contrast, our non-control data gadgets already are syscalls with controllable arguments.

Table 4 presents the number of non-control data gadgets (i.e., syscalls with attacker-tainted arguments) and control data gadgets (i.e., indirect branches with an attacker-tainted branch target) that we identified in each target program, as well as the percentage which have identity flows. Just as previous work does [90], we count indirect branch sites, rather than backtraces; however, counting backtraces give us similar results. We distinguish control data gadgets based on the type of branch target, i.e., whether it is a register or memory operand.

First, we observe that control data gadget with memory operands have higher rates of identity flows than those with register operands. Upon closer inspection, we notice this is because register operands tend be more constrained, as they are typically used for branches with a bounded set of targets (e.g., switch statements), whereas memory operands are typically used for branches with a relatively unbounded set of targets (e.g., indirect calls). Second, we observe that the rates of identity flows for non-control data gadgets and control data gadgets with memory operands are roughly similar. This is unsurprising, because a program treats many types of long-lived data as "immutable", regardless of whether it is control data (e.g., function pointers), or non-control data (e.g., strings). Hence, we conclude that our lightweight identity flow-based analysis is applicable to domains beyond data-only attacks.

C Artifact Appendix

C.1 Abstract

In this artifact, we provide the means to reproduce our main results. Specifically, we show that our exploitation pipeline, EINSTEIN, identifies vulnerable syscalls across a range of applications, and that it generates working data-only exploits against `nginx`. We have evaluated our artifact using an AMD Ryzen 9 3950X CPU (32 cores), with 128GB of RAM, 4 TB storage, and running Ubuntu 22.04.3 LTS (kernel v6.2). Our source code is available on GitHub[5].

[5]https://github.com/vusec/einstein/

C.2 Description & Requirements

C.2.1 Security, privacy, and ethical concerns

Although EINSTEIN indeed produces working exploits, they are non-destructive proof-of-concept exploits, which write the string `"HELLO"` to either a file (`"/tmp/hi"`) or a local socket (address `"192.0.2.0"`). Hence, evaluating EINSTEIN poses no risks for machine security, data privacy, or other ethical concerns.

C.2.2 How to access

The files for the artifact evaluation are available at the `ae` tag of the EINSTEIN repository[6].

C.2.3 Hardware dependencies

EINSTEIN requires an x86-64 machine (Intel recommended); enough RAM to simultaneously load multiple program snapshots into memory, so EINSTEIN can post-process reports in parallel (recommended 100 GB RAM); and enough storage for hundreds of program snapshots (minimum 2 TB storage for this evaluation). We recommend using a machine with a high core count to speed up EINSTEIN's report post-processing.

C.2.4 Software dependencies

To build EINSTEIN and the target programs, we expect certain packages to be installed. In the Section C.3, we detail the steps to install such dependencies on Ubuntu 22.04, but similar steps are needed for other distributions.

C.2.5 Benchmarks

We use each target application's test suite to drive the analysis.

C.3 Set-up

To download and install dependencies, including go-task as a task-runner, from the EINSTEIN repository, run: `sudo snap install task --classic && task init`.

C.3.1 Installation

To build `libdft`[7], the command server, the EINSTEIN tool, and all target applications, run: `task libdft-build cmdsvr-build einstein-build apps-build`.

C.3.2 Basic Test

We first make a couple notes about running EINSTEIN:

- Due to the non-deterministic nature of dynamic analysis (from concurrency issues, system variability, etc.) [17, 33], the actual results may slightly deviate from the expected results.

- If the `db-analyze-reports` task fails, try running the `db-analyze-reports-singleproc` task instead. It will be slower, but will avoid any system load-related crashes.

Test that the different components work as follows:

(T1): *libdft memory tainting [1 compute-second].*
To test libdft's "taint all memory" functionality, run `task libdft-test -- memtaint` *and compare its output to the expected output.*

(T2): *libdft instruction tainting [1 compute-second].*
To test libdft's per-instruction taint policies, run `task libdft-test -- ins` *and compare its output to the expected output.*

(T3): EINSTEIN *tool [1 compute-minute].*
To test EINSTEIN *on a simple program, run* `task einstein-test`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

(T4): *Target applications [4 compute-minutes].*
To test EINSTEIN *running each target application with a simple workload (e.g., sending a simple GET request to a web server), run* `task reports-clean apps-test db-add-reports db-analyze-reports`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

(T5): *Target application test suites [20 compute-minutes].*
To test EINSTEIN *running each target applications' test suites for 2 minutes each (rather than the entire test suites), run* `task reports-clean apps-eval-brief db-add-reports db-analyze-reports`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

(T6): *Exploit confirmation [2 compute-minutes].*
To test EINSTEIN*'s exploit confirmation for* `nginx`*, run* `task reports-clean einstein-nowrite-config nginx-eval-custom db-add-reports db-analyze-reports db-analyze-candidates`*. Then, compare the output of* `task db-print-exploits` *with the expected output.*

C.4 Evaluation workflow

C.4.1 Major Claims

We make the following claims:

(C1): EINSTEIN *identifies thousands of vulnerable syscalls in common server applications. This is proven by Experiment (E1).*

(C2): EINSTEIN *generates hundreds of working exploits against* `nginx`*. This is proven by Experiment (E2).*

C.4.2 Experiments

We prove the above claims using the following experiments:

(E1): *Vulnerable syscall identification [24 compute-hours].*
How to: *We will run each application with* EINSTEIN*, then analyze the reports to identify vulnerable syscalls.*
Preparation: *Run* `task reports-clean` *to remove past reports.*
Execution: *Run* `task apps-eval db-add-reports db-analyze-reports`*.*

[6]https://github.com/vusec/einstein/releases/tag/ae
[7]https://github.com/vusec/libdft64-ng

Results: *Compare the output of* `task db-print-candidates` *to the expected output. The output contains thousands of vulnerable gadgets, broken down by: (i) syscall and argument type (i.e., Table 3), and (ii) target application (i.e., Table 4)—thereby proving Claim (C1).*

(E2): *Exploit generation [12 compute-hours].*

How to: *We will run* `nginx` *with* EINSTEIN, *then analyze the reports to identify vulnerable syscalls, then confirm candidate exploits as working exploits.*

Preparation: *Run* `task reports-clean` *to remove past reports.*

Execution: *Run* `task nginx-eval db-add-reports db-analyze-reports db-analyze-candidates`*.*

Results: *Compare the output of* `task db-print-exploits` *to the expected output. The output contains hundreds of confirmed exploits for* `nginx` *(i.e., Table 5)—thereby proving Claim (C2).*

C.5 Notes on Reusability

This prototype may be expanded in a few directions:

- To modify EINSTEIN's taint policies (e.g, to target more syscalls, or to target syscall-guard variables), modify the EINSTEIN tool in `src/einstein`.

- To run the target applications (e.g., `nginx`) with other workloads, first start the application with EINSTEIN (`cd apps/nginx-1.23.0 && RUN_EINSTEIN=1 ./serverctl restart`), then run the custom workload (e.g., `echo 'Hello!' | netcat 127.0.0.1 1080`).

- To run EINSTEIN on other applications:

  1. Add the application to the `apps/` directory;
  2. Copy the files `serverctl` and `clientctl` from another application's directory into its directory, and modify them to start the application's server and a client for it; and
  3. Ensure that the application's build script generates position-independent code (i.e., the default on most compilers).

- To write another Pin tool that uses `libdft`:

  1. Copy the EINSTEIN tool, e.g.: `cp -r src/einstein src/my-tool`;
  2. Modify `MY_TOOL` and `MY_OBJS` in the `Makefile`;
  3. Modify the source code to suite your analysis;
  4. Build it: `cd src/my-tool && -DLIBDFT_TAG_PTR -DLIBDFT_PTR_32 -DLIBDFT_TAG_SSET_MAX=16' make obj-intel64/my-tool.so`; and
  5. Run it on some target application: `setarch x86_64 -R ./src/misc/pin-3.28-98749-g6643ecee5-gcc-linux/pin -t src/my-tool/obj-intel64/my-tool.so -- echo 'Hello!'`.

C.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.