Automatic Generation of Data-Oriented Exploits

Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang,
National University of Singapore

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-939133-11-3

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

Automatic Generation of Data-Oriented Exploits

Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, Zhenkai Liang
Department of Computer Science, National University of Singapore
{huhong, chuazl, sendroiu, prateeks, liangzk}@comp.nus.edu.sg

Abstract

As defense solutions against control-flow hijacking attacks gain wide deployment, control-oriented exploits from memory errors become difficult. As an alternative, attacks targeting non-control data do not require diverting the application's control flow during an attack. Although it is known that such data-oriented attacks can mount significant damage, no systematic methods to automatically construct them from memory errors have been developed. In this work, we develop a new technique called *data-flow stitching*, which systematically finds ways to join data flows in the program to generate data-oriented exploits. We build a prototype embodying our technique in a tool called FLOWSTITCH that works directly on Windows and Linux binaries. In our experiments, we find that FLOWSTITCH automatically constructs 16 previously unknown and three known data-oriented attacks from eight real-world vulnerable programs. All the automatically-crafted exploits respect fine-grained CFI and DEP constraints, and 10 out of the 19 exploits work with standard ASLR defenses enabled. The constructed exploits can cause significant damage, such as disclosure of sensitive information (e.g., passwords and encryption keys) and escalation of privilege.

1 Introduction

In a memory error exploit, attackers often seek to execute arbitrary malicious code, which gives them the ultimate freedom in perpetrating damage with the victim program's privileges. Such attacks typically hijack the program's control flow by exploiting memory errors. However, such control-oriented attacks, including code-injection and code-reuse attacks, can be thwarted by efficient defense mechanisms such as control-flow integrity (CFI) [10, 43, 44], data execution prevention (DEP) [12], and address space layout randomization (ASLR) [15,33]. Recently, these defenses have become practical and are gaining universal adoption in commodity operating systems and compilers [8, 36], making control-oriented attacks increasingly difficult.

However, control-oriented attacks are not the only malicious consequence of memory error exploits. Memory errors also enable attacks through corrupting non-control data — a well-known result from Chen *et al.* [19]. We refer to the general class of non-control data attacks as *data-oriented attacks*, which allow attackers to tamper with the program's data or cause the program to disclose secret data inadvertently. Several recent high-profile vulnerabilities have highlighted the menace of these attacks. In a recent exploit on Internet Explorer (IE) 10, it has been shown that changing a single byte — specifically the *Safemode* flag — is sufficient to run arbitrary code in the IE process [6]. The Heartbleed vulnerability is another example wherein sensitive data in an SSL-enabled server could be leaked without hijacking the control-flow of the application [7].

If data-oriented attacks can be constructed such that the exploited program follows a legitimate control flow path, they offer a realistic attack mechanism to cause damage even in the presence of state-of-the-art control-flow defenses, such as DEP, CFI and ASLR. However, although data-oriented attacks are conceptually understood, most of the known attacks are straightforward corruption of non-control data. No systematic methods to identify and construct these exploits from memory errors have been developed yet to demonstrate the power of data-oriented attacks. In this work, we study systematic techniques for automatically constructing data-oriented exploits from given memory corruption flaws.

Based on a new concept called *data-flow stitching*, we develop a novel solution that enables us to systematize the understanding and construction of data-oriented attacks. The intuition behind this approach is that non-control data is often far more abundant than control data in a program's memory space; as a result, there exists an opportunity to reuse existing data-flow patterns in the

program to do the attacker's bidding. The main idea of data-flow stitching is to "stitch" existing data-flow paths in the program to form new (unintended) data-flow paths via exploiting memory errors. Data-flow stitching can thus connect two or more data-flow paths that are disjoint in the benign execution of the program. Such a stitched execution, for instance, allows the attacker to write out a secret value (e.g., cryptographic keys) to the program's public output, which otherwise would only be used in private operations of the application.

Problem. Our goal is to check whether a program is exploitable via data-oriented attacks, and if so, to automatically generate working data-oriented exploits. We aim to develop an exploit generation toolkit that can be used in conjunction with a dynamic bug-finding tool. Specifically, from an input that triggers a memory corruption bug in the program, with the knowledge of the program, our toolkit constructs a data-oriented exploit.

Compared to control-oriented attacks, data-oriented attacks are more difficult to carry out, since attackers cannot run malicious code of their choice even after the attack. Though non-control data is abundant in a typical program's memory space, due to the large range of possibilities for memory corruption and their subtle influence on program memory states, identifying how to corrupt memory values for a successful exploit is difficult. The main challenge lies in searching through the large space of memory state configurations, such that the attack exhibits an unintended data consequence, such as information disclosure or privilege escalation. An additional practical challenge is that defenses such as ASLR randomize addresses, making it even harder since absolute address values cannot be used in exploit payloads.

Our Approach. In this work, we develop a novel solution to construct data-oriented exploits through data-flow stitching. Our approach consists of a variety of techniques that stitch data flows in a much more efficient manner compared to manual analysis or brute-force searching. We develop ways to prioritize the searching for data-flow stitches that require a single new edge or a small number of new edges in the new data-flow path. We also develop techniques to address the challenges caused by limited knowledge of memory layout. To further prune the search space, we model the path constraints along the new data-flow path using symbolic execution, and check its feasibility using SMT solvers. This can efficiently prune out memory corruptions that cause the attacker to lose control over the application's execution, like triggering exceptions, failing on compiler-inserted runtime checks, or causing the program to abort abruptly. By addressing these challenges, a data-oriented attack that causes unintended behavior can be constructed, without violating control-flow requirements in the victim program.

We build a tool called FLOWSTITCH embodying these techniques, which operates directly on x86 binaries. FLOWSTITCH takes as input a vulnerable program with a memory error, an input that exploits the memory error, as well as benign inputs to that program. It employs dynamic binary analysis to construct an information-flow graph, and efficiently searches for data flows to be stitched. FLOWSTITCH outputs a working data-oriented exploit that either leaks or tampers with sensitive data.

Results. We show that automatic data-oriented exploit generation is feasible. In our evaluation, we find that multiple data-flow exploits can often be constructed from a single vulnerability. We test FLOWSTITCH on eight real-world vulnerable applications, and FLOWSTITCH automatically constructs 19 data-oriented exploits from eight applications, 16 of which are previously unknown to be feasible from known memory errors. All constructed exploits violate memory safety, but completely respect fine-grained CFI constraints. That is, they create no new edges in the static control-flow graph. All the attacks work with the DEP protection turned on, and 10 exploits (out 19) work even when ASLR is enabled. The majority of known data-oriented attacks (c.f. Chen *et. al.* [19], Heartbleed [7], IE-Safemode [6]) are straightforward non-control data corruption attacks, requiring at most one data-flow edge. In contrast, seven exploits we have constructed are only feasible with the addition of multiple data-flow edges in the data-flow graph, showing the efficacy of our automatic construction techniques.

Contributions. This paper makes the following contributions:

- We conceptualize *data-flow stitching* and develop a new approach that systematizes the construction of data-oriented attacks, by composing the benign data flows in an application via a memory error.
- We build a prototype of our approach in an automatic data-oriented attack generation tool called FLOWSTITCH. FLOWSTITCH operates directly on Windows and Linux x86 binaries.
- We show that constructing data-oriented attacks from common memory errors is feasible, and offers a promising way to bypass many defense mechanisms to control-flow attacks. Specifically, we show that 16 previously unknown and three known data-oriented attacks are feasible from eight vulnerabilities. All our 19 constructed attacks bypass DEP and the CFI checks, and 10 of them bypass ASLR.

2 Problem Definition

2.1 Motivating Example

The following example shown in Code 1 is modeled after a web server. It loads the web site's private key from a

```
1  int server() {
2    char *userInput, *reqFile;
3    char *privKey, *result, output[BUFSIZE];
4    char fullPath[BUFSIZE] = "/path/to/root/";
5
6    privKey = loadPrivKey("/path/to/privKey");
7    /* HTTPS connection using privKey */
8    GetConnection(privKey, ...);
9    userInput = read_socket();
10   if (checkInput(userInput)) {
11     /* user input OK, parse request */
12     reqFile = getFileName(userInput);
13     /* stack buffer overflow */
14     strcat(fullPath, reqFile);
15     result = retrieve(fullPath);
16     sprintf(output,"%s:%s",reqFile,result);
17     sendOut(output);
18   }
19 }
```

Code 1: Vulnerable code snippet. String concatenation on line 14 introduces a stack buffer overflow vulnerability.

file, and uses it to establish an HTTPS connection with the client. After receiving the input — a file name, the code sanitizes the input by invoking `checkInput()` (on line 10). The code then retrieves the file content and sends the content and the file name back to the client. There is a stack buffer overflow vulnerability on line 14, through which the client can corrupt the stack memory immediately after the `fullPath` buffer.

However, there is no obvious security-sensitive non-control data [19] on the stack of the vulnerable function. To create a data-oriented attack, we analyze the data flow patterns in the program's execution under a benign input, which contains at least two data flows: the flow involving the sensitive private key pointed to by the pointer named `privKey`, and the flow involving the input file name pointed by the pointer named `reqFile`, which is written out to the program's public outputs. Note that in the benign run, these two data flows do not intersect — that is, they have no shared variables or direct data dependence between them, but we can corrupt memory in such a way that the secret private key gets written out to the public output. Specifically, the attacker crafts an attack exploiting the buffer overflow to corrupt the pointer `reqFile`, making it to point to the private key. This forces the program to copy the private key to the output buffer in the `sprintf` function on line 16, and then the program sends the output buffer to the client on line 17. Note that the attack alters no control data, and executes the same execution path as the benign run.

This example illustrates the idea of *data-flow stitching*, an exploit mechanism to manipulate the benign data flows in a program execution without changing its control flow. Though it is not difficult to manually analyze this simplified example to construct a data-oriented attack, real-world programs are much more complex and often available in binary-only form. Constructing data-oriented attacks for such programs is a challenging task we tackle in this work.

2.2 Objectives & Threat Model

In this paper, we aim to develop techniques to automatically construct data-oriented attacks by stitching data flows. The generated data-oriented attacks result in the following consequences:

G1: Information disclosure. The attacks leak sensitive data to attackers. Specifically, we target the following sources of security-sensitive data:

- **Passwords and private keys**. Leaking passwords and private keys help bypass authentication controls and break secure channels established by encryption techniques.
- **Randomized values**. Several memory protection defenses utilize randomized values generated by the program at runtime, such as stack canaries, CFI-enforcing tags, and randomized addresses. Disclosure of such information allows attackers bypass randomization-based defenses.

G2: Privilege escalation. The attacks grant attackers the access to privileged application resources. Specifically, we focus on the following kinds of program data:

- **System call parameters**. System calls are used for high-privilege operations, like `setuid()`. Corrupting system call parameters can lead to privilege escalation.
- **Configuration settings**. Program configuration data, especially for server programs (e.g., data loaded from `httpd.conf` for Apache servers) specifies critical information, such as the user's permission and the root directory of the web server. Corrupting such data directly escalates privilege.

Threat Model. We assume the execution environment has deployed defense mechanisms against control-flow hijacking attacks, such as fine-grained CFI [10, 32], non-executable data [12] and state-of-the-art implementation of ASLR. Therefore attackers cannot mount control flow hijacking attacks. All non-deterministic system generated values, e.g., stack-canaries or CFI tags, are assumed to be secret and unknown to attackers.

2.3 Problem Definition

To systematically construct data-oriented exploits, we introduce a new abstraction called the *two-dimensional data-flow graph (2D-DFG)*, which represents the flows of data in a given program execution in two dimensions: memory addresses and execution time. Specifically, a

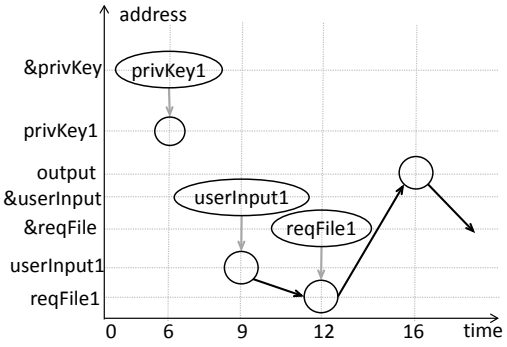

Figure 1: 2D-DFG of a concrete execution of Code 1. Black edges are data edges, while grey edges are address edges. For clarity, vertices do not strictly conform the order on address-axis (this applies to all figures). We use line number to represent the time. *var*1 means a particular value (constant) of the variable `var` in Code 1.

2D-DFG is a directed graph, represented as $G = \{V, E\}$, where V is the set of vertices, and E is the set of edges. A vertex in V is a variable instance, i.e., a point in the two dimensional address-time space, denoted as (a,t), where a is the address of the variable, and t is a representation of the execution time when the variable instance is created. The address includes both memory addresses and register names[1], and the execution time is represented as an instruction counter in the execution trace of the program. An edge (v', v) from vertex v' to vertex v denotes a data dependency created during the execution, i.e., the value of v or the address of v is derived from the value of v'. Therefore, the 2D-DFG also embodies the "points to" relation between pointer variables and pointed variables. Each vertex v has a `value` property, denoted as $v.value$.

A new vertex $v = (a,t)$ is created if an instruction writes to address a at the execution time t. A new data edge (v′, v) is created if an instruction takes v' as the source operand and takes v as a destination operand. A new address edge (v′, v) is created if an instruction takes v′ as the address of one operand v. Therefore, an instruction may create several vertices at a given point in execution if it changes more than one variables, for instance in the loop-prefixed instructions (e.g., `repmov`). Note that the 2D-DFG is a representation of the direct data dependencies created in a program execution under a concrete input, not the static data-flow graph often used in static analysis. Figure 1 shows a 2D-DFG of Code 1.

We define the core problem of *data-flow stitching* as follows. For a program with a memory error, we take the following parameters as the input: a 2D-DFG G from a benign execution of the program, a memory error influence I, and two vertices v_S (source) and v_T (target). In our example, v_S is the private key buffer, shown as (privKey1[2], 6) in Figure 1 and v_T is the public output buffer, shown as (output, 16) in Figure 1. Our goal is to generate an exploit input that exhibits a new 2D-DFG $G' = \{V', E'\}$, where V' and E' result from the memory error exploit, and that G' contains data-flow paths from v_S to v_T. Let $\overline{E} = E' - E$ be the edge-set difference and $\overline{V} = V' - V$ be the vertex-set difference. Then, $\overline{E}$ is the set of new edges we need to generate to get E' from E.

The memory error influence I is the set of memory locations which can be written to by the memory error, represented as a set of vertices. Therefore, we must select $\overline{V}$ to be a subset of vertices in I. To achieve **G1** we consider variables carrying program secrets as source vertices and variables written to public outputs as target vertices. In the development of attacks for **G2**, source vertices are attacker-controlled variables and target vertices are security-critical variables such as system call parameters. A successful data-oriented attack should additionally satisfy the following critical requirements:

- **R1.** The exploit input satisfies the program path constraints to reach the memory error, create new edges and continue the execution to reach the instruction creating v_T.
- **R2.** The instructions executed in the exploit must conform to the program's static control flow graph.

2.4 Key Technique & Challenges

The key idea in data-flow stitching is to efficiently search for the new data-flow edge set $\overline{E}$ to add in G' such that it creates new data-flow paths from v_S to v_T. For each edge $(x,y) \in \overline{E}$, x is data-dependent on v_S and v_T is data-dependent on y. We denote the sub-graph of G containing all the vertices that are data-dependent on v_S as the source flow. We also denote the sub-graph of G containing all the vertices that v_T is data-dependent on as the target flow. For each vertex pair (x, y), where x is in the source flow and y is in the target flow, we check whether (x, y) is a feasible edge of $\overline{E}$ resulting from the inclusion of vertices from I. The vertices x and y may either be contained in I directly, or be connected via a sequence of edges by corruption of their pointers which are in I. If we change the address to which x is written, or change the address from which y is read, the value of x will flow to y. If so, we call (x, y) the *stitch edge*, x the *stitch source*, and y the *stitch target*. For example, in Figure 2, we change the pointer (which is in I) of the file name from *reqFile*1 to *privKey*1. Then the flow of the private key and the flow of the file name are stitched, as we discuss in Section 2.1. In finding data-flow stitching in the 2D-DFG, we face the following challenges:

- **C1. Large search space for stitching.** A 2D-DFG from a real-world program has many data flows and a large number of vertices. For example, there are

[1] We treat the register name as a special memory address.

[2] privKey1 here means the key buffer address, a concrete value.

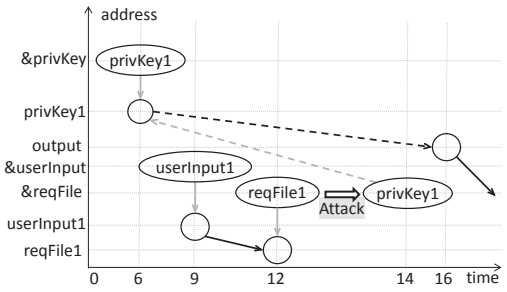

Figure 2: A data-oriented attack of Code 1. This attack connects flow of the private key and flow of the file name, with the new edges (dashed lines).

776 source vertices and 56 target vertices in one of SSHD attacks. Therefore, the search space to find a feasible path is large, for we often need heavy analysis to connect each pair of vertices.

- **C2. Limited knowledge of memory layout.** Most of the modern operating systems have enabled ASLR by default. The base addresses of data memory regions, like the stack and the heap, are randomized and thus are difficult to predict.

The 2D-DFG captures only the data dependencies in the execution, abstracting away control dependence and any conditional constraints the the program imposes along the execution path. To satisfy the requirements **R1** and **R2** completely, the following challenge must be addressed:

- **C3. Complex program path constraints.** A successful data-oriented attack causes the victim program execute to the memory error, create a stitch edge, and continue without crashing. This requires the input to satisfy all path constraints, respect the program's control flow integrity constraints, and avoid invalid memory accesses.

3 Data-flow Stitching

Data-oriented exploits can manipulate data-flow paths in a number of different ways to stitch the source and target vertices. The solution space can be categorized based on the number of new edges added by the exploit. The simplest case of data-oriented exploits is when the exploit adds a single new edge. More complex exploits that use a sequence of corrupted values can be crafted when a single-edge stitch is infeasible. We discuss these cases to solve challenge **C1** in Section 3.1 and 3.2. To overcome the challenge **C2**, we develop two methods to make data-oriented attacks work even when ASLR is deployed, discussed in Section 3.3. For each stitch candidate, we consider the path constraints and CFI requirement (**C3**) to generate input that trigger the stitch edge in Section 4.4.

```
1  struct passwd { uid_t pw_uid; ... } *pw;
2  ...
3  int uid = getuid();
4  pw->pw_uid = uid;
5  ...                  //format string error
6  void passive(void) { ...
7    seteuid(0);              //set root uid
8    ...
9    seteuid(pw->pw_uid);  //set normal uid
10   ... }
```

Code 2: Code snippet of `wu-ftpd`, setting uid back to process user id.

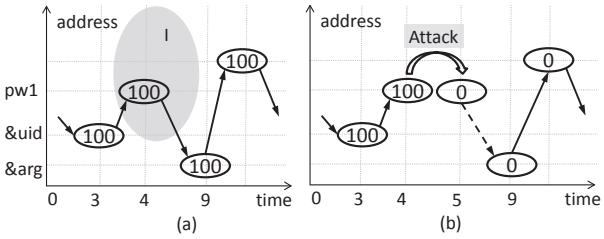

Figure 3: Target flow in the single-edge stitch of `wu-ftpd`. &arg is the stack address of `setuid`'s argument. (a) is the original target flow, where the `pw->pwd_uid` has vale 100 and address *pw*1. Grey area stands for the memory influence *I*. With the stitching attack, the value at address *pw*1 is changed to 0 in (b).

3.1 Basic Stitching Technique

A basic data-oriented exploit adds one edge in the new edge set $\overline{E}$ to connect v_S with v_T. We call this case a *single-edge stitch*. For instance, attackers can create a single new vertex at the memory corruption point by overwriting a security-critical data value, causing escalation of privileges. Most of the previously known data-oriented attacks are cases of single-edge stitches, including attacks studied by Chen *et al.* [19] and the IE Safemode attack [6]. We use the example of a vulnerable web server `wu-ftpd`, shown in Code 2, which was used by Chen *et al.* to explain non-control data attacks [19]. In this exploit, the attackers utilizes a format string vulnerability (skipped on line 5) to overwrite the security-critical `pw->pw_uid` with root user's id. The subsequent `setuid` call on line 9, which is intended to drop the process privileges, instead makes the program retain its root user privileges. Figure 3 (a) and Figure 3 (b) show the 2D-DFG for the execution of the vulnerable code fragment under a benign and the exploit payload respectively. Numbers on time-axis are the line numbers in Code 2. The exploit aims to introduce a single edge to write a zero value from the network input to the memory allocated to the `pw->pw_id`. Note that the exploit is a valid path in the static control-flow graph.

Search for Single-Edge Stitch. Instead of brute-forcing all vertices in the target flow for a stitch edge, we propose a method that utilizes the influence set *I* of the mem-

StitchAlgo-1: Single-edge Stitch

Input: G: benign 2D-DFG, I: memory influence,
v_T: target vertex, cp: memory error vertex,
X: value to be in V_T.value (requirement for stitch edge)
Output: $\overline{E}$: stitch edge candidate set

1 $\overline{E} = \emptyset$
2 $TDFlow$ = dataSubgraph(G, v_T) /* only data edges */
3 **foreach** $v \in V(TDFlow)$ **do**
4 **if** *isRegister(v)* **then**
5 continue /* Skip registers */
6 **if** $\exists\, (v, v') \in E(TDFlow)$: $\exists\, t$: $v.time < t < v'.time \wedge$ $(v.address, t) \in I$ **then**
7 $\overline{E} = \overline{E} \cup \{(cp, v)\}$ /* Stitch edge candidate */

ory error to prune the search space. The influence set I contains vertices that can be corrupted by the memory error, like the grey area shown in Figure 3. For vertices in the target flow, attackers can only affect those in the intersection of the target flow and the influence I. Other vertices do not yield a single-edge stitch and can be filtered out. Specifically, we utilize three observations here. First, register vertices can be ignored since memory error exploit cannot corrupt them. Second, the vertex must be defined (written) before the memory error and used (read) after the memory error. In Figure 3 (a), the code reads vertex (&uid, 3) before the memory error and writes vertices (&arg, 9) and the following one after the memory error. Therefore these three vertices are useless for single-edge stitches. Third, in the memory address dimension, the vertex address should belong to the memory region of the influence I. In our example, only vertex (pw1, 4) falls into the intersection of the target flow and the influence area and we select this vertex for stitch. StitchAlgo-1 shows the algorithm to identify single-edge stitch. From the given 2D-DFG, StitchAlgo-1 gets the target flow $TDFlow$ for the target vertex v_T, which only considers data edges. For each vertex v that satisfies the requirements, we add the edge from memory error vertex to v into $\overline{E}$ as one possible solution.

We consider the search space reduction due to our algorithm over a brute-force search for stitch edges. The naïve brute-force search would consider the Cartesian product of all vertices in the source flow and the target flow. In our algorithm, this search is reduced to the Cartesian product of only the live variables in the source flow at the time of corruption, and the vertices in the target flow as well as in I. In our experiments, we show that this reduction can be significant.

3.2 Advanced Stitching Technique

Single-edge stitch is a basic stitching method, creating one new edge. Advanced data-flow stitching techniques create paths with multiple edges in the new edge set $\overline{E}$. A *multi-edge stitch* can be synthesized through several

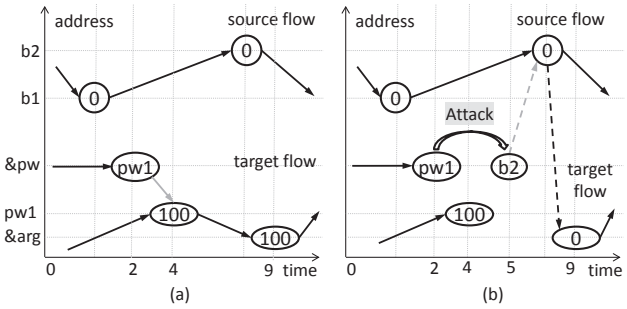

Figure 4: Two-edge stitch of `wu-ftpd`. The target flow is `pw->pw_uid`'s flow, and the source flow is the flow of a constant 0. With the attack, the variable `pw` at `&pw` is changed to $b2$. A later operation reads 0 from $b2$ and writes it to stack for `setuid`. Two edges are changed: one for pointer dereference and another for data movement.

ways. Attackers can use several single-edge stitches to create a multi-edge stitch. Another way is to perform *pointer stitch*, which corrupts a variable that is later used as a pointer to vertices in the source or target flow. Since the pointer determines the address of the stitch source or the stitch target, corrupting the pointer introduces two different edges: one edge for the new "points to" relationship and one edge for the changed data flow. We revisit the example of `wu-ftpd` shown earlier in Code 2, illustrating a multi-edge stitch exploit in it. Instead of directly modifying the field `pw_uid`, we change its base pointer `pw` to an address of a structure with a constant `0` at the offset corresponding to the `pw_uid`. The vulnerable code then reads `0` and uses it as the argument of `setuid`, creating a privilege escalation attack. Figure 4 shows the 2D-DFGs for the benign and attack executions. Changing the value of `pw` creates two new edges (dashed lines): the grey edge that connects the corrupted pointer to a new variable it points to, and the black edge that writes the new variable into `setuid` argument. As a result, we create a two-edge stitch.

Identifying Pointer Stitches. Our algorithm for finding multi-edge exploits using pointer stitching is shown in the StitchAlgo-2. The basic idea is to check each memory vertex in the source flow and the target flow. If it is pointed to by another vertex in the 2D-DFG, we select the pointer vertex to corrupt. The search for stitchable pointers on the target flow is different from that on the source flow. Specifically, for a vertex v in the target flow, we need to find an data edge (v', v) and a pointer vertex vp of v', and then change vp to point to a vertex vs in the source flow, so that a new edge (vs, v) will be created to stitch the data flows. For a vertex v in the source flow, we need to find an data edge (v, v') and a pointer vertex vp of v', and change vp to point to a vertex vt in the target flow, so that a new edge (v, vt) will be created to stitch the data flows. At the same time, we need to consider the liveness of the stitching vertices. For example, the source vertex

StitchAlgo-2: Pointer Stitch

Input: G: benign 2D-DFG, I: memory influence,
v_S: source vertex, v_T: target vertex,
cp: memory error vertex
Output: $\overline{E}$: stitch edge candidate set

```
1  Ē = ∅
2  SrcFlow = subgraph(G, v_S)        /* both data and address edges. */
3  TgtFlow = subgraph(G, v_T)
4  SDFlow = dataSubgraph(G, v_S)     /* only data edges */
5  TDFlow = dataSubgraph(G, v_T)
6  foreach v ∈ V(TDFlow) do
7      if isRegister(v) then continue
8      if ∄ (vi ∈ E(I) ∧ (v, v') ∈ TDFlow) : vi.time < v'.time then
       continue
9      foreach (vp, v) ∈ E(TgtFlow) – E(TDFlow) do
           /* Only consider address edges.                    */
10         if vp is used to write v then continue
           /* Expect data flow from v                         */
11         foreach vs ∈ V(SDFlow) do
12             if ¬isRegister(vs) ∧ vs.isAliveAt(vp.time) then
13                 StitchAlgo-1(G, I, vp, cp, vs.address)
14 foreach v ∈ V(SDFlow) do
15     if isRegister(v) then continue
16     if ∀ vi ∈ I: v.time < vi.time then continue
17     foreach (vp, v) ∈ E(SrcFlow) – E(SDFlow) do
18         if vp is used to read v then continue
           /* Expect data flow into v                         */
19         foreach vt ∈ V(TDFlow) do
20             if ¬ isRegister(vt) ∧∃(vt, v') ∈ TDFlow : vt.time
               < vp.time < v'.time then
21                 StitchAlgo-1(G, I, vp, cp, vt.address)
```

should carry valid source data when it is used to write data out to the target vertex. Once we select the pointer vertex vp and its value (vt's or vs's address), the last step is to set the value into vp through the memory error exploit. StitchAlgo-2 invokes the basic stitching technique in StitchAlgo-1 to complete the last step.

Our technique uses vertex liveness and the memory error influence I to significantly reduce the search space. A naïve solution to finding pointer stitches would consider all pairs (vs, vt) where vs is in the source flow and vt is in the target flow. The search space will be the Cartesian product of the vertex set in the source flow (denoted as $V(SrcFlow)$) and the vertex set in the target flow (denoted as $V(TgtFlow)$). In contrast, in StitchAlgo-2, if the memory corruption occurs at time $t1$, the vertex used in the stitch edge from the source flow must be live at $t1$. Similarly, the vertex used in the stitch edge from the target flow should be created after $t1$. We illustrate it in Figure 5, where only the black vertices are candidates. Furthermore, we restrict our search to the set of vertices whose pointer vertices vp are inside the memory influence as well. We call the selected vertices from the source flow *R-set*. Similarly, we call the vertices selected from the target flow *W-set*. Our algorithm reduces

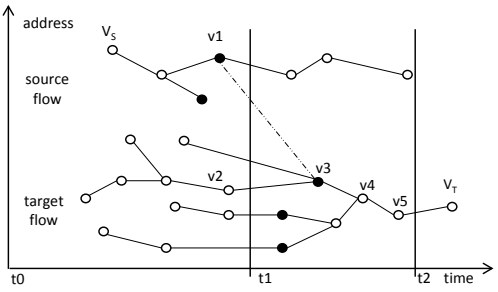

Figure 5: Stitch edge selection. The execution starts at time $t0$, and reaches memory error instructions at time $t1$. Target data is used at time $t2$, just before target vertex V_T. There are two stitch source candidates (black points in the source flow) and three stitch destination candidate (black points in the target flow). One of the stitch edge candidate is shown using the dotted line.

the search space to the Cartesian product of the *R*-set and *W*-set instead.

$$R\text{-}set = V(SrcFlow) \cap I, \quad W\text{-}set = V(TgtFlow) \cap I$$

$$|SS_{naive}| = |V(SrcFlow)| \times |V(TgtFlow)|$$
$$|SS_{pointer-stitch}| = |R\text{-}set| \times |W\text{-}set|$$

Pointer stitch constitutes a natural hierarchy of exploits, which can consist of multiple levels of dereferences of attacker-controlled pointers. For instance, in a *two-level pointer stitch* we can construct an exploit that corrupts a pointer vp_2 that points to the pointer vp. This can be achieved by treating vp as the target vertex, another pointer vp' holding the intended value (vt's or vs's address) as the source vertex and applying StitchAlgo-2 to change vp. In this case, StitchAlgo-2 is recursively used twice. Similarly, *N-level stitch* corrupts a pointer vp_N of the the pointer $vp_{(N-1)}$ to make an attack (and so on), by applying StitchAlgo-2 N times recursively. Note that for a N-level stitch to work, we need to make sure the source vertex vp'_N "aligns" with the target vertex vp_N at each level, such that the program dereferences vp_N N-1 times to get the vertex vp, and dereferences vp'_N N-1 time to get the intended value in the exploit.

Pointer stitch is one specific way to implement multi-edge stitches. In principle, it can be composed to create more powerful exploits, combining several other single-edge stitches in a "multi-step" stitch attack. In a multi-step stitch, several intermediate data flows are used to achieve data-flow stitching. Each step can be realized by pointer stitch or single-edge stitch. Multi-step stitch is useful when direct stitches of the source flow and the target flow are not feasible.

3.3 Challenges from ASLR

Address space layout randomization (ASLR) deployed by modern systems poses a strong challenge in mounting

Table 1: Deterministic memory region size of binaries on Ubuntu 12.04 x86 system. Position-independent executables have size 0. Two largest numbers are highlighted for each directory.

size (KB)	/bin	/sbin	/usr/bin	/usr/sbin	Total
0	21	22	73	18	134
1 - 8	10	**33**	**150**	**20**	**213**
8 - 16	12	17	113	11	153
16 - 32	**23**	17	**147**	14	**201**
32 - 64	**19**	**22**	103	**25**	169
64 - 128	15	8	66	8	97
128 - 256	7	2	35	4	48
256 - 512	3	2	32	3	40
> 512	2	2	32	2	38
Total	112	125	751	105	1093

successful data-oriented attacks since vertex addresses are highly unpredictable. We develop two methods in data-oriented attacks to address this challenge: stitching with deterministic addresses and stitching by address reuse. Note that attackers can use others methods developed for control flow attacks to bypass ASLR here, like disclosure of random addresses [14, 35].

3.3.1 Stitching With Deterministic Addresses

When security-critical data is stored in deterministic memory addresses, stitching data flows of such data is not affected by ASLR. Existing work [2, 34, 37] have shown that current ASLR implementations leave a large portion of program data in the deterministic memory region. For example, Linux binaries are often compiled without the "*-pie*" option, resulting in deterministic memory regions. We study deterministic memory size of Ubuntu 12.04 (x86) binaries under directories `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin`, and show the results in Table 1. Among 1093 analyzed programs, more than 87.74% have deterministic memory regions. Two hundred and twenty-three programs have deterministic memory regions larger than 64KB. Inside such memory regions, there is many security-critical data, like randomized addresses in `.got.plt` and configuration structures in `.bss`. Hence we believe stitch with deterministic addresses in real-world programs is practical.

We build an information leakage attack against the `orzhttpd` web server [5] (details in Section 6.4.3) using the stitch with deterministic addresses. To respond to a page request, `orzhttpd` uses a pointer to retrieve the HTTP protocol version string. The pointer is stored in memory. If we replace the pointer value with the address of a secret data, the server will send that secret to the client. However this requires both the address of the pointer and the address of the secret to be predictable. In the `orzhttpd` example, we find that the address of the pointer is fixed (`0x8051164`) and choose the contents of the `.got.plt` section (allocated at a fixed address) as the secret to leak out. Figure 6 shows two 2D-

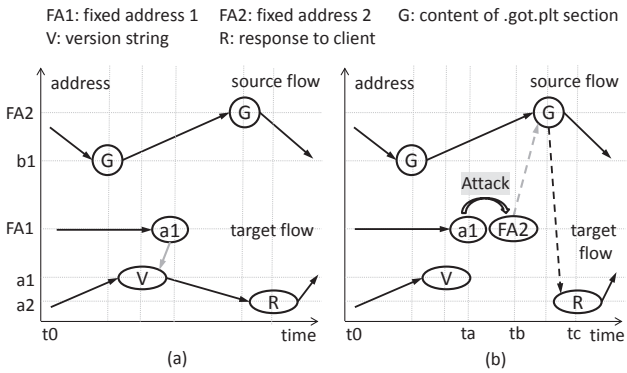

Figure 6: Stitch with deterministic memory addresses of the `orzhttpd` server. This attack is the similar to the one in Figure 4, except the address of the source vertex and the pointer's address of of the target vertex are fixed. This attack works with ASLR.

DFGs for the benign execution and the attack, respectively. With this attack, the content of `.got.plt` is sent to the attacker, which leads to an memory address disclosure exploit useful for constructing second-stage control-hijacking attacks or stealing secret data in randomized memory region. Unlike a direct memory disclosure attack, here we use the corruption of deterministically-allocated data to leak randomized addresses.

Identifying Stitch with Deterministic Addresses. We represent the deterministic memory region as a set *D*. Our algorithm considers the intersection of *D* for the vertices in the source flow and the target flow. The previously outlined stitching algorithms can then be used directly prioritizing the vertices in the intersection with *D*.

3.3.2 Stitching By Address Reuse

If the security-critical data only exists inside the randomized memory region, data-oriented attacks cannot use deterministic addresses. To bypass ASLR in such cases, we leverage the observation that a lot of randomized addresses are stored in memory. If we can reuse such real-time randomized addresses instead of providing concrete address in the exploit, the generated data-oriented attacks will be stable (agnostic to address randomization). There are two types of address reuse: partial address reuse and complete address reuse.

Partial Address Reuse. A variable's relative address, with respect to the module base address or with respect to another variable in the same module, is usually fixed. Attackers can easily calculate such relative addresses in advance. On the other hand, instructions commonly get a memory address with one base address and one relative offset (e.g., array access, switch table). If attackers control the offset variable, they can corrupt the offset with the pre-computed relative address from the selected vertex (source vertex or target vertex) and reuse the randomized base address. In this way attackers can access

```
1  struct user_details { uid_t uid; ... } ud;
2  ...                   //run with root uid
3  ud.uid = getuid();    //in get_user_info()
4  ...
5  vfprintf(...);        //in sudo_debug()
6  ...
7  setuid(ud.uid);       //in sudo_askpass()
8  ...
```

Code 3: Code snippet of `sudo`, setting uid to normal user id.

the intended data without knowing their randomized addresses. We show an example of a vulnerable instruction pattern, that allows the attacker partial ability to read a value from memory and write it out without knowing randomized addresses. If attackers control `%eax`, they can reuse the source base address `%esi` in the first instruction, and reuse the destination base address `%edi` in the second instruction. In fact, any memory access instruction with a corrupted offset can be used to mount partial address reuse attack.

```
1  //attackers control %eax
2  mov  (%esi,%eax,4), %ebx  //reuse %esi
3  mov  %ecx, (%edi,%eax,4)  //reuse %edi
```

Complete Address Reuse. We observe that a variable's address is frequently saved in memory due to the limitation of CPU registers. If the memory error allows retrieving such spilled memory address for reading or writing, attackers can reuse the randomized vertex address existing in memory to bypass ASLR. For example, in the following assembly code, if attacker controls `%eax` on line 1, it can load a randomized address into `%ebx` from memory. Then, attacker can access the target vertex pointed by `%ebx` without knowing the concrete randomized address. The attacker merely needs to know the right offset value to use in `%eax` on line 2, or may have a deterministic `%esi` value to gain arbitrary control over addresses loaded on line 2.

```
1  //attacker controls %eax
2  mov  (%esi, %eax, 4), %ebx
3  mov  %ecx, (%ebx) / mov (%ebx), %ecx
```

Let us consider a real example of the `sudo` program [9] that shows how to use such instruction patterns that permit complete address reuse meaningfully. Code 3 shows the related code of `sudo`, where a format string vulnerable exists in the `sudo_debug` function (line 5). At the time of executing `vfprintf()` on line 5, the address of the user identity variable (`ud.uid`) exists on the stack. The `vfprintf()` function with format string "*%X*$n" uses the *X*th argument on stack for "%n". By specifying the value of *X*, `vfprintf()` can retrieve the address of `ud.uid` from its ancestor's stack frame and change the `ud.uid` to the root user ID without knowing

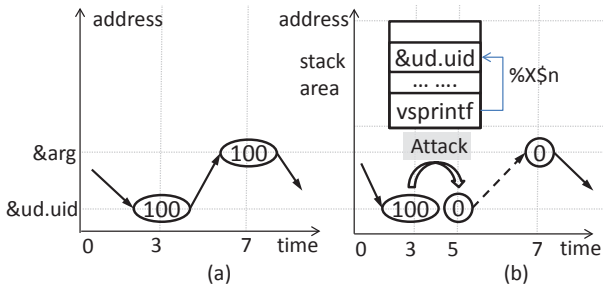

Figure 7: Stitch by complete memory address reuse of `sudo`. The dashed line is the new edge (single-edge stitch). An address of *ud.uid* exists on ancestor's stack frame, which is reused to overwrite *ud.uid*.

the stack base address. Figure 7 shows the 2D-DFGs for the benign execution and the attack. This attack works even if the fine-grained ASLR is deployed.

Identifying Stitch by Address Reuse. Memory error instructions for address reuse stitch should match the patterns we discuss above. For partial address reuse, the memory error exploit corrupts variable offsets, while for complete address reuse, the memory error exploit can retrieve addresses from memory. Our approach intersects the memory error influence *I* with the source flow and the target flow. Then we search from the new source flow and the new target flow to identify matched instructions, from which we can build stitch by address reuse with methods discuss above.

4 The FLOWSTITCH System

We design a system called FLOWSTITCH to systematically generate data-oriented attacks using data-flow stitching. As shown in Figure 8, FLOWSTITCH takes three inputs: a program with memory errors, an error-exhibiting input, and a benign input of the program. The two inputs should drive the program execution down the same execution path until the memory error instruction, with the error-exhibiting input causing a crash. FLOWSTITCH builds data-oriented attacks using the memory errors in five steps. First, it generates the execution trace for the given program. We call the execution trace with the benign input the *benign trace*, and the execution trace with the error-exhibiting input the *error-exhibiting trace*. Second, FLOWSTITCH identifies the influence of the memory errors from the error-exhibiting trace and generates constraints on the program input to reach memory errors. Third, FLOWSTITCH performs data-flow analysis and security-sensitive data identification using the benign trace. Fourth, FLOWSTITCH selects stitch candidates from the identified security-sensitive data flows with the methods discussed in Section 3. Finally, FLOWSTITCH checks the feasibility of creating new edges with the memory errors and validates the exploit. It finally outputs the input to mount a data-oriented attack.

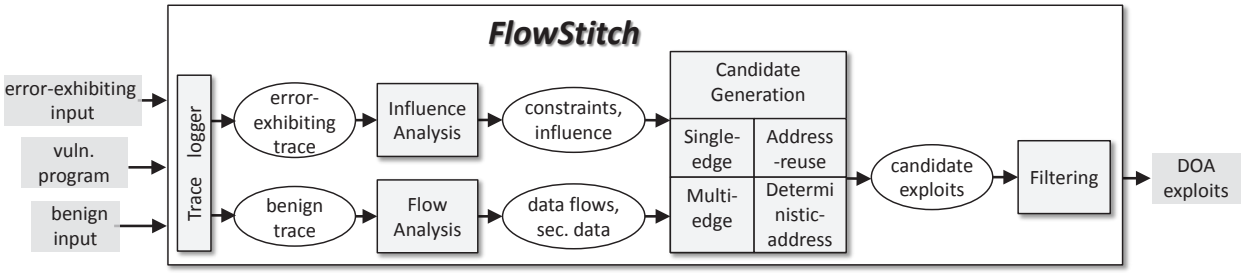

Figure 8: Overview of FLOWSTITCH. FLOWSTITCH takes a vulnerable program, an error-exhibiting input and a benign input of the program as inputs. It builds data-oriented attacks against the given program using data-flow stitching. Finally it outputs the data-oriented attack exploits.

FLOWSTITCH requires that the error-exhibiting input and the benign input make the program follow the same code path until memory error happens. Such pairs of inputs can be found by existing symbolic execution tools, like BAP [16] and SAGE [25], which explore multiple execution paths with various inputs. Before detecting one error-exhibiting execution, these tools usually have explored many matched benign executions.

4.1 Memory Error Influence Analysis

FLOWSTITCH analyzes the error-exhibiting trace to understand the influence *I* of the memory errors. It identifies two aspects of the memory error influence: the time when the memory errors happens during the execution (temporal influence) and the memory range that can be written to in the memory error (spatial influence). From the error-exhibiting trace, FLOWSTITCH detects instructions whose memory dereference addresses are derived from the error-exhibiting input. We call these instructions *memory error instructions*. Note that data flows ending before such instructions or starting after them cannot be affected by the memory error, therefore they are out of the temporal influence.

Attackers get access to unintended memory locations with memory error instructions. However, the program's logic limits the total memory range accessible to attackers. To identify the spatial influence of the memory error instruction, we employ dynamic symbolic execution techniques. We generate a symbolic formula from the error-exhibiting trace in which all the inputs are symbolic variables and all the path constraints are asserted true. Inputs that satisfy the formula imply that the execution to memory error instructions with an unintended address[3]. The set of addresses that satisfy these constraints and can be dereferenced at the memory error instruction constitute the spatial influence.

[3]This is true if the symbolic formula constructed is complete [26].

4.2 Security-Sensitive Data Identification

As we discuss in Section 2.3, FLOWSTITCH synthesizes flows of security-sensitive data. There are four types of data that are interesting for stitching: input data, output data, program secret and permission flags. To identify input data, FLOWSTITCH performs taint analysis at the time of trace generation, treating the given input as an external taint source. For output data, FLOWSTITCH identifies a set of program sinks that send out the program data, like `send()` and `printf()`. The parameters used in sinks are the output data. Further, we classify program secret and permission flags into two categories: the program-specific data and the generic data. FLOWSTITCH accepts user specification to find out program-specific data. For example, user can provide addresses of security flags. For the generic data, FLOWSTITCH uses the following methods to automatically infer it.

- **System call parameters.** FLOWSTITCH identifies all system calls from the trace, like `setuid`, `unlink`. Based on the system call convention, FLOWSTITCH collects the system call parameters.
- **Configuration data.** To identify configuration data, FLOWSTITCH treats the configuration file as a taint source and uses taint analysis to track the usage of the configuration data.
- **Randomized data.** FLOWSTITCH identifies stack canary based on the instructions that set and check the canary, and identifies randomized addresses if they are not inside the deterministic memory region.

Deterministic Memory Region Identification. FLOWSTITCH identifies the deterministic memory region for stitch with deterministic addresses (Section 3.3.1). It first checks the program binary to identify the memory regions that will not be randomized at runtime. If the program is not position-independent, all the data sections shown in the binary headers will be at deterministic addresses. FLOWSTITCH collects loadable sections and gets a deterministic memory set *D*. FLOWSTITCH further scans benign traces to find all the memory writing instructions that write data into the deterministic memory set to identify data stored in such region.

Note that based on the functionality of the security-sensitive data, we predefine goals of the attacks. For example, the attack of `setuid` parameter is to change it to the root user's id 0. For a web server's home directory string, the goal is to set it to system root directory.

4.3 Stitching Candidate Selection

For identified security-sensitive data, FLOWSTITCH generates its data flow from the 2D-DFG. FLOWSTITCH selects the source flow originated from the source vertex V_S and the target flow ended at the target vertex V_T. It then uses the stitching methods discussed in Section 3 to find stitching solutions. Although any combination of stitching methods can be used here, FLOWSTITCH uses the following policy in order to produce a successful stitching efficiently.

1. FLOWSTITCH tries the single-edge stitch technique before the multi-edge stitch technique. After the single-edge stitch's search space is exhausted, it moves to multi-edge stitch. FLOWSTITCH stops searching at four-edge stitch in our experiments.
2. FLOWSTITCH considers stitch with deterministic addresses before stitch by address reuse. After exhausting the search space of deterministic address and address reuse space, FLOWSTITCH continues searching stitches with concrete addresses shown in benign traces, for cases without ASLR.

4.4 Candidate Filtering

To overcome challenge **C3**, FLOWSTITCH checks the feasibility of each selected stitch edge candidate. We define the *stitchability constraint* to cover the following constraints.

- Path conditions to reach memory error instructions;
- Path conditions to continue to the target flow;
- Integrity of the control data;

FLOWSTITCH generates the stitchability constraint using symbolic execution tools. The constraint is sent to SMT solvers as an input. If the solver cannot find any input satisfying the constraint, FLOWSTITCH picks the next candidate stitch edge. If it exists, the input will be the witness input that is used to exercise the execution path in order to exhibit the data-oriented attack. Due to the concretization in symbolic constraint generation in the implementation, the constraints might not be complete [26], i.e., it may allow inputs that results in different paths. FLOWSTITCH concretely verifies the input generated by the SMT solver to check if it successfully mounts the data-oriented attack on the program.

5 Implementation

We prototype FLOWSTITCH on Ubuntu 12.04 32 bit system. Note that as the first step the trace generation tool can work on both Windows and Linux systems to generate traces. Although the following analysis steps are performed on Ubuntu, FLOWSTITCH works for both Windows and Linux binaries.

Trace Generation. Our trace generation is based on the Pintraces tool provided by BAP [16]. Pintraces is a Pin [28] tool that uses dynamic binary instrumentation to record the program execution status. It logs all the instructions executed by the program into the trace file, together with the operand information. In our evaluation, the traces also contain dynamic taint information to facilitate the extraction of data flows.

Data Flow Generation. For input data and configuration data, FLOWSTITCH uses the taint information to get the data flows. To generate the data flow of the security-sensitive data, FLOWSTITCH performs backward and forward slicing on the benign trace to locate all the related instructions. It is possible for one instruction to have multiple source operands. For example, in `add %eax, %ebx`, the destination operand `%ebx` is derived from `%eax` and `%ebx`. In this case, one vertex has multiple parent vertices. As a result, the generated data flow is a graph where each node may have multiple parents.

Constraint Generation and Solving. The generation of the stitchability constraint required in Section 4.4 is implemented in three parts: path constraints, influence constraints, and CFI constraints.The stitchability constraint is expressed as a logical conjunction of these three parts. We use BAP to generate formulas which capture the path conditions and influence constraints. For control flow integrity constraint, we implement a procedure to search the trace for all the indirect `jmp` or `ret` instruction. Memory locations holding the return addresses or indirect jump targets are recorded. The control flow integrity requires that at runtime, the memory location containing control data should not be corrupted by the memory errors. The stitchability constraint is checked for satisfiability using the Z3 SMT-solver [22], which produces a witness input when the constraint is satisfiable.

6 Evaluation

In this section, we evaluate the effectiveness of data-flow stitching using FLOWSTITCH, including single-edge stitch, multi-edge stitch, stitch with deterministic addresses and stitch by address reuse. We also measure the search space reduction using FLOWSTITCH and the performance of FLOWSTITCH.

6.1 Efficacy in Exploit Generation

Table 2 shows the programs used in our evaluation, as well as their running environments and vulnerabilities. The trace generation phase is performed on different

Table 2: Experiment environments and benchmarks. # of Data-Oriented Attacks gives the number of attacks generated by FLOWSTITCH, including privilege escalation attacks and information leakage attacks. FLOWSTITCH generates 19 data-oriented attacks from 8 vulnerable programs.

ID	Vul. Program	Vulnerability	Environment (32b)	# of Data-Oriented Attacks	
				Escalation	Leakage
CVE-2013-2028	nginx	stack buffer overflow	Ubuntu 12.04	1	1
CVE-2012-0809	sudo	format string	Ubuntu 12.04	1	0
CVE-2009-4769	httpdx	format string	Windows XP SP3	4	1
bugtraq ID: 41956	orzhttpd	format string	Ubuntu 9.10	1	1
CVE-2002-1496	null httpd	heap overflow	Ubuntu 9.10	2	0
CVE-2001-0820	ghttpd	stack buffer overflow	Ubuntu 12.04	1	0
CVE-2001-0144	SSHD	integer overflow	Ubuntu 9.10	2	1
CVE-2000-0573	wu-ftpd	format string	Ubuntu 9.10	2	1
Total	8 programs			14	5

Table 3: Evaluation of FLOWSTITCH on generating data-oriented attacks. In the Attack Description column, L_i stands for information leakage attack, while M_i represents privilege escalation attack. The third column indicates whether the built attack can bypass ASLR or not. The "CP" column shows the number of memory error instructions. Trace size is the number of instructions inside the trace. The last four columns show the number of stitch sources and stitch targets before and after our selection. SrcFlow means source flow, while TgtFlow stands for target flow.

Vul. Apps	Attack Description	ASLR Bypass	CP	Error-exhibiting Trace Size	Benign Trace Size	# of nodes before		# of nodes after	
						SrcFlow	TgtFlow	SrcFlow	TgtFlow
nginx	L_0: private key		1	50789	411437	3	48	3	1
	M_0: http directory path				1717182	173	462	1	42
sudo	M_0: user id	✓	1	351988	854371	2083	1	1	1
httpdx	L_0: admin's password	✓	1	1197657	1361761	152	7	152	2
	M_0: admin's password	✓			1298247	78	120	1	8
	M_1: anon.'s permission	✓			1233522	78	2	1	1
	M_2: anon.'s root directory	✓			1522672	78	165	1	11
	M_3: CGI directory path	✓			1257694	78	480	1	30
orzhttpd	L_0: randomized address	✓	1	84694	131871	8	28	8	1
	M_0: directory path	✓			131871	368	95	1	19
null httpd	M_0: http directory path		2	160844	401285	3	141	2	47
	M_1: CGI directory path				335329	3	144	2	48
ghttpd	M_0: CGI directory path		1	312130	316473	3579	6	1	1
SSHD	L_0: root password hash		1	38201	3094592	776	56	97	2
	M_0: user id				674365	1	24	1	1
	M_1: authenticated flag				674365	1	2	1	1
wu-ftpd	L_0: env. variables		1	328108	1417908	88	5	88	1
	M_0: user id (single-edge)	✓			1057554	183	2	1	1
	M_1: user id (multi-edge)	✓			1057554	183	1	1	1

systems according to the tested program. All generated traces are analyzed by FLOWSTITCH on a 32-bit Ubuntu 12.04 system. The vulnerabilities used for the experiments come from four different categories to ensure that FLOWSTITCH can handle different vulnerabilities. Seven of the 8 vulnerable programs are server programs, including HTTP and FTP servers, which are the common targets of remote attacks. The other one is the `sudo` program, which allows users to run command as another user on Unix-like system. The last four vulnerabilities were discussed in [19], where data-oriented attacks were manually built. We apply FLOWSTITCH on these vulnerabilities to verify the efficacy of our method.

Results. Our result demonstrates that FLOWSTITCH can effectively generate data-oriented attacks with different vulnerabilities on different platforms. The number of generated data-oriented attacks on each program is shown in Table 2 and their details are given in Table 3. FLOWSTITCH generates a total of 19 data-oriented attacks for eight real-world vulnerable programs, more than two attacks per program on average. Among 19 data-oriented attacks, there are five information leakage attacks and 14 privilege escalation attacks. For the vulnerable `httpdx` server, FLOWSTITCH generates five data-oriented attacks from a format string vulnerability.

Out of the 19 data-oriented attacks, 16 are previously unknown. The three known attacks are two `uid`-corruption attacks on `SSHD` and `wu-ftpd`, and a CGI directory corruption attack on `null httpd`, discussed in [19]. FLOWSTITCH successfully reproduces known attacks and builds new data-oriented attacks with the same vulnerabilities. Note that FLOWSTITCH produces a different `ghttpd` CGI directory corruption attack than the one described in [19]. Details of this attack are discussed in Section 6.4.2. The results show the efficacy of our systematic approach in identifying new data-oriented attacks.

From our experiments, seven out of 19 of the data-

oriented attacks are generated using multi-edge stitch. The significant number of new data-oriented attacks generated by multi-edge stitch highlights the importance of a systematic approach in managing the complexity and identifying new data-oriented attacks. As a measurement of the efficacy of ASLR on data-oriented attacks, we report that 10 of 19 attacks work even with ASLR deployed. Among 10 attacks, two attacks reuse randomized addresses on the stack and eight attacks corrupt data in the deterministic memory region. We observe that security-sensitive data such as configuration option is usually represented as a global variable in C programs and reside in the `.bss` segment. This highlights the limitation of current ASLR implementations which randomize the stack and heap addresses but not the `.bss` segment.

For three of 19 attacks, FLOWSTITCH requires the user to specify the security-sensitive data, including the private key of `nginx`, the root password hash and the authenticated flag of `SSHD`. For others, FLOWSTITCH automatically infers the security-sensitive data using techniques discussed in Section 4.2. Once such data is identified, FLOWSTITCH automatically generates data-oriented exploits.

6.2 Reduction in Search Space

Data-flow stitching has a large search space due to the large number of vertices in the flows to be stitched. Manual checking through a large search space is difficult. For example, in the root password hash leakage attack against `SSHD` server, there are 776 vertices in source flow containing the hashed root passwords. In the target flow, there are 56 vertices leading to the output data. Without considering the influence of the memory errors, there are a total of 43,456 possible stitch edges. After applying the methods described in Section 3, we get the intersection of the memory error influence I with the stitch source set R-set and the stitch target set W-set. In this way, the number of candidate edges is reduced from 43,456 to 194, obtaining a reduction ratio of 224.

The last four columns in Table 3 give the detailed information of the search space for each attack. For most of the data-oriented attacks, there is a significant reduction in the number of possible stitches. `ghttpd`-M_0 achieves the highest reduction ratio of 21,474 while `SSHD`-M_1 achieves the lowest reduction ratio of two. The median reduction ratio is 183 achieved by `wu-ftpd`-M_1(multi-edge). Given the relatively large spatial influence of the memory error, most of the reduction is achieved by the temporal influence of I.

6.3 Performance

We measure the time FLOWSTITCH uses to generate data-oriented attacks. Table 4 shows the results, including the time of trace generation and the time of data-flow collection (slicing). Note that the trace generation time includes the time to execute instructions that are not logged (e.g., `crypto` routines and `mpz` library for `SSHD`). As we can see from Table 4, FLOWSTITCH takes an average of six minutes and 27 seconds to generate the trace and flows. Most of them are generated within 10 minutes. The information leakage attack of `SSHD` server takes the longest time, 34 minutes and 23 seconds, since crypto routines execute a large number of instructions. From the performance results, we can see that the generation of data flows through trace slicing takes up most of the generation time, from 20 percent to 87 percent. Currently, our slicer works on BAP IL file. We plan to optimize the slicer using parallel tools in the future.

Table 4: Performance of trace and flow generation using FLOWSTITCH. The unit used in the table is second, so 1:07 means one minute and seven seconds.

Attacks		Trace Gen		Slicing		Total
		error	benign	error	benign	
nginx	L_0	0:08	0:22	0:06	2:41	3:17
	M_0		0:36		0:12	1:02
sudo	M_0	0:35	1:07	1:17	3:34	6:33
httpdx	L_0	0:08	0:45	0:12	5:56	7:01
	M_0		0:51		4:44	5:55
	M_1		0:50		4:52	6:02
	M_2		1:03		4:45	6:08
	M_3		0:53		4:47	6:00
orzhttpd	L_0	0:17	0:20	0:12	0:24	1:13
	M_0		0:20		1:04	1:53
null httpd	M_0	0:13	1:20	0:14	6:21	8:08
	M_1		0:52		2:29	3:48
ghttpd	M_0	0:09	0:18	0:12	0:09	0:48
SSHD	L_0	2:35	9:38	1:02	21:08	34:23
	M_0		5:30		1:22	10:29
	M_1		5:30		1:00	10:07
wu-ftpd	L_0	0:12	0:50	0:19	5:42	7:03
	M_0		0:31		0:27	1:29
	M_1		0:31		0:26	1:28
Average		0:32	1:41	0:26	3:47	6:27

6.4 Case Studies

We present five case studies to demonstrate the effectiveness of stitching methods and interesting observations.

6.4.1 Sensitive Data Lifespan

A common defense employed to reduce the effectiveness of data-oriented attacks is to limit the lifespan of security-critical data [19, 20]. This case study highlights the difficulty of doing it correctly. In the implementation of `SSHD`, the program explicitly zeros out sensitive data, such as the RSA private keys, as soon as they are not in use. For password authentication on Linux, `getspnam()` provided by glibc is often used to obtain the password hash. Rather than using the password hash directly, `SSHD` makes a local copy of the password

hash on stack for its use. Although the program makes no special effort is to clear the copy on the stack, the password on stack is eventually overwritten by subsequent function frames before it can be leaked. The developer explicitly deallocates the original hash value using `endspent()` [1] in the glibc internal data structures. However, glibc does not clear the deallocated memory after `endspent()` is called and this allows FLOWSTITCH to successfully leak the hash from the copy held by glibc. Hence, this case study highlights that sensitive information should not be kept by the program after usage, and that identifying all copies of sensitive data in memory is difficult at the source level.

6.4.2 Multi-edge Stitch – ghttpd CGI Directory

The `ghttpd` application is a light-weight web server supporting CGI. A stack buffer overflow vulnerability was reported in version 1.4.0 - 1.4.3, allowing remote attackers to smash the stack of the vulnerable `Log()` function. During the security-sensitive data identification, FLOWSTITCH detects `execv()` is used to run an executable file. One of `execv()`'s arguments is the address of the program path string. Controlling it allows attackers to run arbitrary commands. FLOWSTITCH is unable to find a new data dependency edge using single-edge stitching, since there is no security-sensitive data on the stack frame to corrupt. FLOWSTITCH then proceeds to search for a multi-edge stitch. For the program path parameter of `execv()`, FLOWSTITCH identifies its flow, which includes use of a series of stack frame-base pointers saved in memory. The temporal constraints of the memory error exploit only allow the saved `%ebp` of the `Log()` function to be corrupted. Once the `Log()` function returns, the saved `%ebp` is used as a pointer, referring to all the local variables and parameters of `Log()` caller's stack frame. FLOWSTITCH corrupts the saved `%ebp` to change the variable for the CGI directory used in `execv()` system call. This attack is a four-edge stitch by composing two pointer stitches.

Chen *et al.* [19] discussed a data-oriented attack with the same vulnerability, which was in fact a two-edge stitch. However, that attack no longer works in our experiment. The `ghttpd` program compiled on our Ubuntu 12.04 platform does not store the address of command string on the stack frame of `Log()`. Only the four-edge stitching can be used to attack our `ghttpd` binary.

6.4.3 Bypassing ASLR – orzhttpd Attacks

The `orzhttpd` web server has a format string vulnerability which the attacker can exploit to control almost the whole memory space of the vulnerable program. FLOWSTITCH identifies the deterministic memory region and the randomized address on stack under `fprintf()` frame. The first attack which bypasses ASLR is a privilege escalation attack. This attack corrupts the web root directory with single-edge stitching and memory address reuse. The root directory string is stored on the heap, which is allocated at runtime. FLOWSTITCH identifies the address of the heap string from the stack and reuses it to directly change the string to `/` based on the pre-defined goal (Section 4.2). The second attack is an information leakage attack, which leaks randomized addresses in the `.got.plt` section. FLOWSTITCH identifies the deterministic memory region from the binary and performs a multi-edge stitch. The stitch involves modifying the pointer of an HTTP protocol string stored in a deterministic memory region. FLOWSTITCH changes the pointer value to the address of `.got.plt` section and a subsequent call to send the HTTP protocol string leaks the randomized addresses to attackers.

6.4.4 Privilege Escalation – Nginx Root Directory

The `Nginx` HTTP server 1.3.9-1.4.0 has a buffer overflow vulnerability [4]. FLOWSTITCH checks the local variables on the vulnerable stack and identifies two data pointers that can be used to perform arbitrary memory corruption. The memory influence of the overwriting is limited by the program logic. FLOWSTITCH identifies the web root directory string from the configuration data. It tries single-edge stitching to corrupt the root directory setting. The root directory string is inside the memory influence of the arbitrary overwriting. FLOWSTITCH overwrites the value `0x002f` into the string location, thus changing the root directory into `/`. FLOWSTITCH verifies the attack by requesting `/etc/passwd` file. As a result, the server sends the file content back to the client.

6.4.5 Information Leakage – httpdx Password

The `httpdx` server has a format string vulnerability between version 1.4 to 1.5 [3]. The vulnerable `tolog()` function records FTP commands and HTTP requests into a server-side log file. Note that direct exploitation of this vulnerability does not leak information. Using the error-exhibiting trace, FLOWSTITCH identifies the memory error instruction and figures out that there is almost no limitation on the memory range affected by attackers. From the `httpdx` binary, FLOWSTITCH manages to find a total of 102MB of deterministic memory addresses. From the benign trace, FLOWSTITCH generates data flows of the root user passwords. This is the secret to be leaked out. The FLOWSTITCH generates the necessary data flow which reaches the `send()` system call automatically.

Starting from the memory error instruction, FLOWSTITCH searches backwards in the secret data flow and identifies vertices inside the deterministic memory region. FLOWSTITCH successfully finds two such memory locations containing the "admin" password: one is a

buffer containing the whole configuration file, and another only contains the password. At the same time, FLOWSTITCH searches forwards in the output flow to find the vertices that affect the buffer argument of `send()`. Our tool identifies vertices within the deterministic memory region. The solver gives one possible input that will trigger the attack. FLOWSTITCH confirms this attack by providing the attack input to the server and receiving the "admin" user password.

7 Related Work

Data-Oriented Attack. Several work [21, 32, 36, 38, 41, 43, 44] has been done to improve the practicality of CFI, increasing the barrier to constructing control flow hijacking attacks. Instead, data-oriented attacks are serious alternatives. Data-oriented attacks have been conceptually known for a decade. Chen *et al.* constructed non-control-data exploits to show that data-oriented attack is a realistic threat [19]. However, no systematic method to develop data-oriented attacks is known yet. In our paper, we develop a systematic way to search for possible data-oriented attacks. This method searches attacks within the candidate space efficiently and effectively.

Automatic Exploit Generation. Brumley *et al.* [17] described an automatic exploit generation technique based on program patches. The idea is to identify the difference between the patched and the unpatched binaries, and generate an input to trigger the difference. Avgerinos *et al.* [13] discussed Automatic Exploit Generation(AEG) to generate real exploits resulting in a working shell. Felmetsger *et al.* [24] discussed automatic exploit generation for web applications. The previous work focused on generating control flow hijacking exploits. FLOWSTITCH on the other hand generates data-oriented attacks that do not violate the control flow integrity. To our knowledge, FLOWSTITCH is the first tool to systematically generate data-oriented attacks.

Defenses against Data-Oriented Attacks. Data-oriented attacks can be prevented by enforcing data-flow integrity (DFI). Existing work enforces DFI through dynamic information tracking [23, 39, 40] or by legitimate memory modification instruction analysis [18, 42]. However, DFI defenses are not yet practical, requiring large overheads or manual declassification. An ultimate defense is to enforce the memory safety to prevent the attacks in their first steps. Cyclone [27] and CCured [31] introduce a safe type system to the type-unsafe C languages. SoftBound [29] with CETS [30] uses bound checking with fat-pointer to force a complete memory safety. Cling [11] enforces temporal memory safety through type-safe memory reuse. Data-oriented attack prevention requires a complete memory safety.

8 Conclusion

In this paper, we present a new concept called data-flow stitching, and develop a novel solution to systematically construct data-oriented attacks. We discuss novel stitching methods, including single-edge stitch, multi-edge stitch, stitch with deterministic addresses and stitch by address reuse. We build a prototype of data-flow stitching, called FLOWSTITCH. FLOWSTITCH generates 19 data-oriented attacks from eight vulnerable programs. Sixteen attacks are previously unknown attacks. All attacks bypass DEP and the CFI checks, and 10 bypass ASLR. The result shows that automatic generation of data-oriented exploits exhibiting significant damage is practical.

Acknowledgments. We thank R. Sekar, Shweta Shinde, Yaoqi Jia, Xiaolei Li, Shruti Tople, Pratik Soni and the anonymous reviewers for their insightful comments. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate, and in part by a research grant from Symantec.

References

[1] Endspent(3C). `https://docs.oracle.com/cd/E36784_01/html/E36874/endspent-3c.html`.

[2] How Effective is ASLR on Linux Systems? `http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/`.

[3] HTTPDX tolog() Function Format String Vulnerability. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4769`.

[4] Nginx HTTP Server 1.3.9-1.4.0 Chunked Encoding Stack Buffer Overflow. `http://mailman.nginx.org/pipermail/nginx-announce/2013/000112.html`.

[5] OrzHTTPd. `https://code.google.com/p/orzhttpd/`.

[6] Subverting without EIP. `http://mallocat.com/subverting-without-eip/`.

[7] The Heartbleed Bug. `http://heartbleed.com/`.

[8] Visual Studio 2015 Preview: Work-in-Progress Security Feature. `http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx`.

[9] Sudo Format String Vulnerability. `http://www.sudo.ws/sudo/alerts/sudo_debug.html`, 2012.

[10] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005).

[11] AKRITIDIS, P. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium* (2010).

[12] ANDERSEN, S., AND ABELLA, V. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory protection technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004.

[13] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. AEG: Automatic Exploit Generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium* (2011).

[14] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security* (2014).

[15] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium* (2003).

[16] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011).

[17] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy* (2008).

[18] CASTRO, M., COSTA, M., AND HARRIS, T. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006).

[19] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium* (2005).

[20] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 14th USENIX Security Symposium* (2005).

[21] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014).

[22] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[23] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010).

[24] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Security Symposium* (2010).

[25] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium* (2008), Internet Society.

[26] GODEFROID, P., AND TALY, A. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012).

[27] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference* (2002).

[28] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005).

[29] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIG-PLAN Conference on Programming Language Design and Implementation* (2009).

[30] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 9th International Symposium on Memory Management* (2010).

[31] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2002).

[32] NIU, B., AND TAN, G. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014).

[33] PAX TEAM. PaX Address Space Layout Randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[34] PAYER, M., AND GROSS, T. R. String Oriented Programming: When ASLR is Not Enough. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop* (2013).

[35] SERNA, F. J. The Info Leak Era on Software Exploitation. *Black Hat USA* (2012).

[36] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, U., LOZANO, L., AND PIKE, G. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[37] UBUNTU. List of Programs Built with PIE, May 2012. `https://wiki.ubuntu.com/Security/Features#pie`.

[38] WANG, Z., AND JIANG, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy* (2010).

[39] XU, W., BHATKAR, S., AND SEKAR, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium* (2006).

[40] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009).

[41] ZENG, B., TAN, G., AND ERLINGSSON, U. Strato: A Retargetable Framework for Low-level Inlined-reference Monitors. In *Proceedings of the 22nd USENIX Security Symposium* (2013).

[42] ZENG, B., TAN, G., AND MORRISETT, G. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (2011).

[43] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (2013).

[44] ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium* (2013).