A2C: Self Destructing Exploit Executions via Input Perturbation

Yonghwi Kwon*, Brendan Saltaformaggio*, I Luk Kim*, Kyu Hyung Lee†, Xiangyu Zhang*, and Dongyan Xu*
*Department of Computer Science, Purdue University
{kwon58, bsaltafo, kim1634, xyzhang, dxu}@cs.purdue.edu
†Department of Computer Science, University of Georgia
kyuhlee@cs.uga.edu

***Abstract*—Malicious payload injection attacks have been a serious threat to software for decades. Unfortunately, protection against these attacks remains challenging due to the ever increasing diversity and sophistication of payload injection and triggering mechanisms used by adversaries. In this paper, we develop A2C, a system that provides general protection against payload injection attacks. A2C is based on the observation that payloads are highly fragile and thus any mutation would likely break their functionalities. Therefore, A2C mutates inputs from untrusted sources. Malicious payloads that reside in these inputs are hence mutated and broken. To assure that the program continues to function correctly when benign inputs are provided, A2C divides the state space into exploitable and post-exploitable sub-spaces, where the latter is much larger than the former, and decodes the mutated values only when they are transmitted from the former to the latter. A2C does not rely on any knowledge of malicious payloads or their injection and triggering mechanisms. Hence, its protection is general. We evaluate A2C with 30 real-world applications, including `apache` on a real-world work-load, and our results show that A2C effectively prevents a variety of payload injection attacks on these programs with reasonably low overhead (6.94%).**

I. Introduction

Attacks which exploit software vulnerabilities are among the most prevalent cyber-security threats to date. This is due, in part, to many complex combinations of potential attack vectors: Buffer overflow attacks, Return-to-libc attacks [58], ROP [50], Jump-oriented programming (JOP) [10], and Heap spraying [60], [27] to name just a few. Unfortunately, this ever expanding variety of exploit attack vectors has led to a constant "cat and mouse game" of building defenses as each new attack is released.

In light of this, many existing protection mechanisms focus on specific attack vectors and become less effective (or even completely ineffective) for others. For example, non-executable stack and heap have difficulty preventing code reuse (e.g., ROP) attacks because the executable payload is constructed from the original code of the application. Shellcode detection techniques are only effective against injection of binary executable code and are often bypassable [32], [26], [39], [65]. Control Flow Integrity [66], [31], [73], [43] prevents attacks which exhibit certain abnormal control flows within a victim program. Further, some defense techniques may entail non-trivial overhead (e.g., [6]) or require hardware support (e.g., [41]), which affects their application in practice. Based on this trend of *attack-specific defense*, we are motivated to look for an entirely new, more fundamental weakness of software exploits to provide an *attack vector independent* protection mechanism.

It turns out that all software exploit attacks invariably have two common characteristics: First, they all need to inject an exploit payload into the target application. This payload could be a piece of executable code (shellcode) or information that allows constructing the malicious instruction sequence at runtime (e.g., a ROP chain that contains the entry addresses of gadgets). Second, these payloads are famously *brittle*. Specifically, exploit payloads are designed with very strict semantic assumptions about the environment (e.g., memory layout, libraries, or known binary instructions) which require *each byte of the payload to be carefully tailored to a victim.*

In this paper, we will show that these invariant characteristics of exploit attacks make it possible to protect applications from exploit injections *independent* of the attack vector they use. Specifically, we leverage the observation that *exploit payloads (regardless of their attack vector) are so brittle that any mutation would break their execution* — i.e., cause the execution to crash. For example, even simple mutation of x86 shellcode results in invalid instructions. Similarly, most sequences of ROP addresses no longer form an executable gadget chain if even a single byte is changed. Secondly, since these exploit payloads must be injected into a victim application, their behavior eventually diverges from that of the application's legitimate inputs. Therefore, we propose that exploit payloads may be easily disabled via a "shoot first and ask questions later" policy, whereby all input to a victim program is immediately mutated and only those that are beyond the control of the adversary are decoded.

Based on the above observations, we have developed the A2C (or "Attack to Crash") technique. A2C naturally exploits the brittleness of attack payloads by setting these attacks on track to crash before malicious logic is executed. First, any buffer inputs from untrusted sources are securely encoded using A2C's *One-Time Dictionary*, which varies for each input buffer to prevent memory disclosure/value guessing based attacks. Since all the untrusted inputs are mutated, malicious payloads that reside in these inputs are also mutated, resulting in broken payloads which will induce crashes when executed. Later, A2C must undo the mutation in the buffer inputs, when the program begins using these inputs to compute new values, so that our mutation does not cause any exceptions for legitimate input.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23476

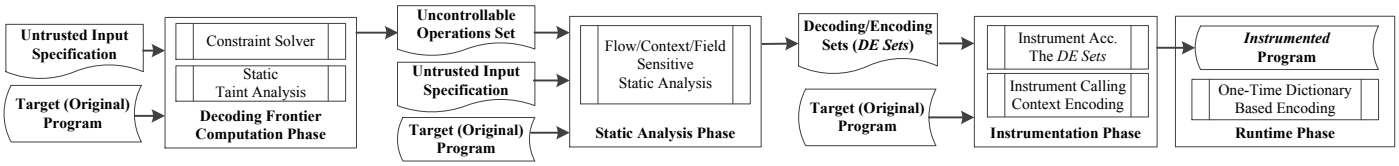

Fig. 1. Overall procedure of A2C.

Our evaluation shows that A2C is able to protect a variety of applications against a wide spectrum of exploit attacks regardless of their injection methods, without affecting the normal functionalities of the program. Further, because A2C requires no knowledge of the specific attacks (only leveraging the two invariant characteristics mentioned above) it may even prevent currently unknown injection attack types in the future. The detailed threat model considered in this paper is presented in Section V.

Our contributions are summarized in the following:

- We propose the novel idea of partitioning program state space into the *exploitable* and *post-exploitable* sub-spaces so that we only need to protect the smaller exploitable sub-space, which is critical to A2C's efficiency and effectiveness.
- We develop a novel constraint solving based approach that can determine the boundary of the two sub-spaces. This serves as the basis to compute the execution points where the mutation can be safely undone.
- We develop a flow-, context-, and field-sensitive static analysis to identify the places at which A2C needs to undo the mutation so that execution on legitimate inputs is not affected.
- We develop an efficient runtime that leverages a *One-Time Dictionary*, which projects a value to another unique value. The dictionary varies for each input buffer to prevent memory disclosure based attacks. A2C also features efficient calling context encoding to support undoing input mutation.
- We develop a prototype A2C. The evaluation results show that A2C effectively prevents a number of known payload injection attacks with low overhead (6.94%).

II. System Overview

In this section, we present an overview of A2C, which is based on the following two observations. (1) *Most malicious payloads reside in buffers and they only go through copy operations or simple transformations before the attack is launched.* It is very rare for these payloads to undergo complex transformations in the victim program before being executed. This is due to the difficulty in controlling the transformations (in the victim program) to generate meaningful payloads. (2) *Malicious payloads are very fragile. Any mutation often leads to an unsuccessful attack.* For example, changing a few bits at the beginning of a shellcode can easily throw off the sequence of executed instructions, leading to a crash.

The overarching idea of A2C is to *protect a program from malicious injection attacks by perturbing or encoding inputs from untrusted sources*. However, inputs from untrusted sources (e.g., packets from remote IPs) are not necessarily malicious. We need to ensure that our perturbation does not fail executions based on non-exploit inputs. According to observation (1), we aim to undo the perturbation when the buffer

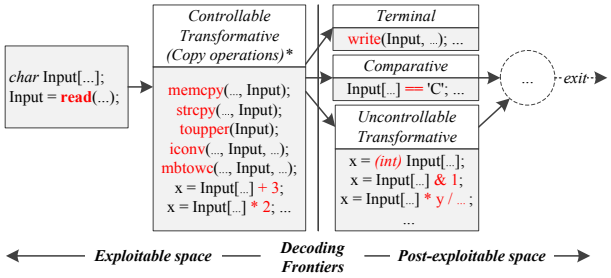

Fig. 2. Decoding frontiers.

data goes beyond copy operations/simple transformations and starts being used in benign computation.

In the following, we use the diagram in Fig. 2 to illustrate the life cycle of buffer data and hence the intuition behind A2C. After the buffer data are loaded through input functions, they may undergo a number of transformations, including copy operations (e.g., `memcpy()` and `strcpy()`) that copy a buffer to another target buffer, constant table lookup (e.g., in `iconv()`, `toupper()`, `mbtowc()`, and `wctomb()`), and simple transformative operations (e.g., additions with a constant). Then, the buffer data will eventually encounter one of the following three kinds of operations: (1) *Comparative operations*, in which elements in the buffer are used in comparisons; (2) *Terminal operations*, in which the buffer data are passed to output library functions (e.g., `write()`, `send()`, and `printf()`); (3) *Uncontrollable transformative operations*, in which elements in the buffer undergo transformations that disallow the attacker to control the values beyond these transformations to construct meaningful payloads. For instance, *type widening* copies a value of smaller type (e.g., char) to an array element of larger type (e.g., integer) so that each element in the array is padded with leading 0's. As such, the resulting byte sequence denoted by the array cannot serve as a meaningful payload.

We call these three kinds of operations the *decoding frontier* (DF) because A2C should undo the perturbation for the buffer elements involved before executing the operations. Intuitively, we consider the space before the frontier the *exploitable space* where the malicious payloads are supposed to take effect and *without* perturbation would successfully exploit the program. Therefore, we use perturbation to achieve protection in this space. The space after the frontier is referred to as the *post-exploitable space*. This is because controlling the payload becomes infeasible if it has gone through these benign transformations conducted by the victim program. Therefore, it is safe to undo our perturbation before the decoding frontier so that benign inputs can be used in computation as usual[1]. The core technical challenge for A2C is hence to identify the DF of a subject program and perform instrumentation accordingly. More discussion about the decoding frontier can be found in Section IV-A.

Another interesting observation that makes our solution

[1]Here we assume that output library functions are hardened and thus cannot be exploited by the decoded buffers.

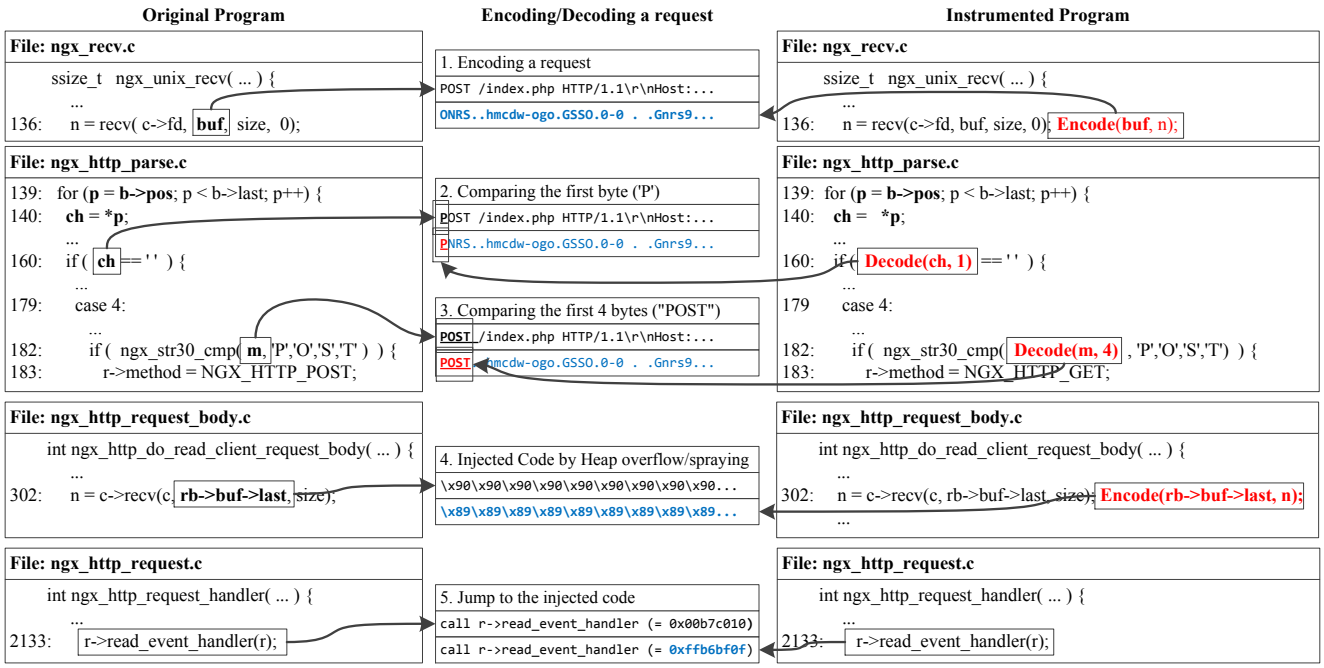

Fig. 3. Original and instrumented programs of demonstrative example.

feasible is that the exploitable space is usually much smaller than the post-exploitable space as most computation happens in the post-exploitable space. As such, the frontier tends to be small and shallow and as explained above, operations beyond the frontier do not need our attention.

Overall Procedure. Fig. 1 shows the complete procedure of A2C. There are four phases: *constraint solving based decoding frontier computation*, *static analysis for determining encoding and decoding places which are a superset of the decoding frontier*, *instrumentation*, and *runtime*.

First, we leverage constraint solving to determine the uncontrollable operations. These operations, together with the comparative and terminal operations, form the decoding frontier. This phase simply marks all the operations on the frontier.

Second, a flow-, context-, and field-sensitive analysis is applied to determine the places to instrument. It takes three inputs: the LLVM IR of the program, the decoding frontier from the first phase, and the untrusted input specification that identifies a set of library functions that read inputs, such as `recv()` for network inputs and `read()` for file streams. In this phase, A2C produces two outputs. Specifically, the *decoding set* is a superset of the decoding frontier and the *encoding set* contains the statements to encode (input) values, such as `recv()` in network programs. Interestingly, the encoding set may also contain instructions that load constant values. Explanations about why we need to encode constants can be found in Section IV-C. The computation of decoding and encoding sets (DE sets for short) is iterative as new elements on encoding sets may introduce additional decoding operations.

Third, the instrumentation phase statically instruments the program according to the DE sets. An important observation is that the decoding frontier is context sensitive. Different inputs may lead to different calling contexts of a function invocation. The membership of a statement in the DE set may change with those contexts. As such, upon the execution of a statement in the DE set, we need to know the current calling context to determine if the instrumented version or the original version of the statement should be executed. Therefore, part of the instrumentation phase handles the problem of efficiently tracking the current calling context.

Lastly, the runtime supports execution of the instrumented program. It features encoding based on a *One-Time Dictionary*, which projects a plaintext value to a unique encoded value. Different input buffers use different dictionaries to prevent memory exposure based exploits.

III. Illustrative Example

In this section, we use a real-world example to illustrate A2C's operation. We use the `nginx` 1.4.0 web-server as the subject program. It has two known heap buffer overflow and integer overflow vulnerabilities, which can be triggered by providing crafted HTTP requests containing malicious payloads. Fig. 3 shows two code snippets with part of the original `nginx` program on the left and the corresponding instrumented version on the right. The column in the middle shows how the two code snippets process the request differently.

First, both programs receive a POST request at Line 136 in `ngx_recv.c`. Since the request is from an untrusted source, the instrumented program encodes the buffer. For simplicity of discussion, the encoding here is to subtract 1 from every byte. `Encode()` denotes this modification. The HTTP request "`POST /index.php HTTP/1.1\r\nHost:...`" is hence encoded as "`ONRS..hmcdw-ogo.GSSO.0-0..Gnrs9...`". The request is parsed at Lines 160 and 182 in `ngx_http_parse.c`, which contain comparative operations on some buffer data and are hence part of the decoding frontier. Therefore, the instrumented program calls `Decode()` to undo the perturbation so that the program can parse and process the request correctly. Note that it only decodes a few bytes (of fixed length) at a time so that *the decoded data cannot be run as any meaningful payload*. Also observe that the original buffer remains encoded. This is achieved by only decoding the

values after they are loaded into variables of primitive types (e.g., bytes and words).

Next, the `ngx_http_do_read_client_request_body()` function stores the contents of the request into a different heap buffer. Notice that without A2C this becomes vulnerable to heap spraying attacks which can be further leveraged to launch attacks such as ROP. Also, the same function has a heap buffer overflow vulnerability that allows overwriting a function pointer, `read_event_handler`, which will be called inside `ngx_http_request_handler()`. However, since the instrumented program encodes all external requests, the payload at Line 302 and the address accessed at Line 2133 are mutated. Assume the malicious shellcode contains a sequence of nop instructions (`0x90*n`) for the *nop-sled* portion of a heap spray attack and the malicious address injected is `0x00b7c010`. In the instrumented program, the nop instructions (`0x90*n`) are encoded to "`0x89*n`", which denotes a sequence of mov instructions that write to invalid memory locations (e.g. `mov ecx, ecx(-76767677h)`). At this point, even though the shellcode is successfully injected, due to the mutation, it crashes upon execution. Similarly, the injected function pointer at Line 2133 is also broken. Note that if the request is valid, despite it being encoded by the instrumented program, it will be decoded at the frontier and will not affect normal execution.

IV. Design

A. Decoding Frontier Computation via Constraint Solving.

The first phase of A2C is to determine the decoding frontier that will be used to identify the encoding and decoding sets in the next analysis phase. As we will see in the next section, A2C needs to decode at more places than input related buffers.

According to the definition in Section II, the decoding frontier consists of three kinds of operations: comparative, terminal, and uncontrollable. While the identification of the first two is straightforward, we focus on the third in this section.

We first define *controllable operations* as follows: *if valid payloads can be generated in a memory region (e.g., a buffer) right after a set of operations by manipulating program inputs, these operations are controllable*. An example of a controllable operation is the `toupper()` transformation that turns a lower case character into its upper case. Assume an application transforms a text input buffer A into another buffer B using `toupper()`. The attacker can carefully prepare the input so that after the transformation, buffer B contains the intended payload. It was indeed reported that existing operations in a program could be leveraged to compute/decode payloads [5].

We further formulate the determination of controllable operations as a constraint solving problem. We consider program inputs as symbolic variables. We further model the operations that compute the values for a memory region (at a given program point) from the program inputs as a set of constraints. We then assert the values (of the memory region) to be some valid payload and query a solver if there is a satisfying (SAT) solution. If so, one may be able to manipulate the input (e.g., using the SAT solution generated by the solver) to induce the given payload. While it is difficult to precisely define what constitutes a valid payload, we use the following procedure to determine if operations are controllable.

Procedure to Determine Decoding Frontier. Given a program to protect, A2C identifies all memory regions larger than or equal to 16 bytes that can be affected by inputs (through a standard static taint analysis). These regions include buffers, consecutive local variables (on stack), consecutive global variables (in data section), as well as structures. For example, four consecutive local integer variables related to inputs constitute a region for testing. For these regions, A2C creates constraints according to the operations that compute the values in the regions from program inputs. Other variables that are not related to inputs are considered as free variables. This makes our analysis a conservative one as free variables can take any values during constraint solving, whereas in practice these variables may have various restrictions. After we generate the constraints, we use the Z3 solver [25] to test whether payloads can be generated through these operations. In particular, we collected 1.4GB binary codes, 200MB shellcode, and 200MB ROP gadgets from Internet [1], [3], [51], [53], [52]. We also generate 1.0G random numbers. We further break the data sets down to sequences based on the size of the region under testing. If the size is unknown, we use 16-byte sequences. We then assert the values of the region equal to each of these sequences one by one. If the constraint solver yields SAT, TIMEOUT, or UNKNOWN for *any* of the sequences, which implies that an attacker may be able to construct some malicious payload through the operations, then the operations are considered controllable. If the constraints are UNSAT for *all* these sequences, the operations that define the values of the memory region are considered uncontrollable.

Essence. Intuitively, we use the large pool of binary code and shell code snippets to model the distribution of executable payloads and the large pool of ROP gadget subsequences to model the distribution of address-based payloads (for code reuse attacks). We further use a large set of random number sequences to model the distribution of other *arbitrary* payloads. Since we only consider operations uncontrollable when *all* these sequences yield UNSAT results, A2C provides strong probabilistic guarantees that the values beyond these operations are not exploitable.

Note that for complex programs, it may be difficult to model the entire data flow from program inputs to the memory region of interest due to various reasons such as unmodeled library calls and uncertainty of data flow caused by aliasing. A2C leverages backward slicing, starting from the memory region of interest and traverses backward along data dependencies until the traversal becomes infeasible (e.g., due to unmodeled library calls). If program inputs cannot be reached by the traversal, A2C treats the farthest variables that it can reach as free variables. Note that this yields an over-approximation, which is safe. The decoding frontier analysis marks all the operations on the decoding frontier. Since the algorithms in this phase are standard, details are omitted.

In the following, we use a number of examples to facilitate understanding of decoding frontier.

Uncontrollable Operation Example One. Fig. 4 shows a code snippet from 464.h264ref (i.e., a video decoding program) in SPEC 2006.

(a) Code snippet from 464.h264ref **(b) Constraints from the code snippet**

```
// Declarations (Data Types)                 ; Constraints for Operations (img - mpr)
1. unsigned int    m7[...][...];             7.  m7[0,1,2,3] = img[0,1,2,3] - mpr[0,1,2,3]   ∧
2. unsigned short  img[...][...];
3. unsigned short  mpr[...][...];            ; Constraints for the range of unsigned short
   ...                                       8.  0 <= img[0,1,2,3] ∧ 0 <= mpr[0,1,2,3]       ∧
// Transformative Operations                 9.  img[0,1,2,3] <= 65535 ∧ mpr[0,1,2,3] <= 65535 ∧
4. for (int x = 0; ...; x++ )
5.   for (int y = 0; ...; y++ )              ; Constraints for Payloads (i will select a payload)
6.     m7[x][y] = img[...][...] - mpr[...][...];  10. m7[0,1,2,3] = payload[i, i+1, i+2, i+3]
```

Fig. 4. Uncontrollable operations due to type widening in 464.h264ref.

Fig. 4 (a) shows three arrays `m7`, `img`, and `mpr` with `m7` a temporary array that stores intermediate values during encoding, `img` holding raw input values and `mpr` calculated by the program and not related to inputs. Observe that `m7` is an *int* array whereas the other two are arrays of *short int*. Fig. 4 (b) shows the constraints generated. Lines 7-9 denote the constraints representing the operations. Line 7 denotes the subtraction at Line 6. Line 9 denotes the range constraints of `img` and `mpr`. We use "0,1,2,3" to represent that the same constraint applies to four respective elements. Line 9 denotes the payload assertion. We iterate this test with i from 0 to the number of sequences in our test data set.

The test result shows that the constraints are always UNSAT. This is mainly because the assignment of *short* to *int* (called *type widening*) requires payloads to have two zero bytes in every four bytes. As such, Line 6 is on the decoding frontier. Type widening is one of the major reasons for uncontrollability. Another popular form of type widening is through bit operations, namely, only a few bits of a word are set. Examples are omitted.

Uncontrollable Operation Example Two. Another common kind of uncontrollable operation is one that induces intensive correlations between values. For example, Fig. 5 (a) shows a code snippet from 429.mcf in SPEC.

(a) Code snippet from 429.mcf **(b) Constraints from the code snippet**

```
// Declaration (Data Types)                  ; Constraints for Operations
1. typedef struct network{                   11. net[0] = (2 * in[0] + 1)            ∧
2.   long  n, n_trips, max_m, m;             12. net[1] = in[0]                      ∧
       ...                                   13. ( (net[2] = (3 * in[0] + in[1])) ∨
3. } network_t;                              14.   (net[2] = 0xA10001))              ∧
       ...                                   15. net[3] = (3 * in[0] + in[1])        ∧
4.   network_t* net;
5.   in[2] = read( InputFile );             ; Constraints for Payloads
// Transformative Operations                 ; (i will select a payload to test)
6.   net->n_trips = in[0];                   16. net[0] = payload[i]      ∧
       ...                                   17. net[1] = payload[i+1]    ∧
7.   net->n = (in[0]+in[0]+1);               18. net[2] = payload[i+2]    ∧
8.   net->m = (in[0]+in[0]+in[0]+in[1]);     19. net[3] = payload[i+3]
9.   if ( ... ) net->max_m = net-> m;
10. else     net->max_m = 0xA10001;
```

Fig. 5. Uncontrollable operations in 429.mcf program.

Fields `n`, `n_trips`, `max_m`, and `m` are consecutive in the structure `network` and they are all related to inputs (`in[0]` and `in[1]`). As such, A2C needs to test if the operations on these fields are controllable. The constraints are shown in Fig. 5 (b). Observe that the `net→max_m` (i.e., `net[3]` in the constraint) and `net→m` (i.e., `net[4]`) are identical except when `net→max_m` has a constant value `0xA10001`. The other 8 bytes are also closely correlated through `in[0]` and `in[1]`. Consequently, the solver returns UNSAT for all the payload tests.

Controllable Operation Examples. Most controllable operations are straightforward, such as copy operations. Method `toupper()` is another example of a controllable operation. The solver returns SAT for many payload sequences, such as consecutive 0x90's, which represent the NOP instructions (nop-sled) in exploits. A2C also determines unicode conversion functions (e.g., `mbtowc()`) as controllable. This is because while unicode conversion translates an ASCII character to two bytes with an additional byte (0x00), it also translates two byte characters such as Chinese, Japanese, and Korean characters to two bytes [63], making payload construction feasible. Our results echo the message conveyed in [5] that Unicode conversion function can be leveraged to construct payloads. In fact, all the data conversion/encryption/decryption/encoding via table lookup (e.g., `iconv()`, `mbtowc()`, `wctomb()`, and Inflate (Huffman Coding) Algorithm) are recognized as controllable by A2C.

(a) Code snippet from 456.hmmer **(b) Constraints from the code snippet**

```
// Declarations (Data Types)                 ; Constraints for Operations
1. float v[...], sum;                        10. sum = v_old[0] + v_old[1] + v_old[2] + v_old[3] ∧
2. int x, n;                                 11. (v_new[0] = (v_old[0] / sum) or (1.0 / n)) ∧
// Transformative Operations                 12. (v_new[1] = (v_old[1] / sum) or (1.0 / n)) ∧
3.  sum = FSum(v, n);                        13. (v_new[2] = (v_old[2] / sum) or (1.0 / n)) ∧
    // FSum returns a sum of all elements.   14. (v_new[3] = (v_old[3] / sum) or (1.0 / n)) ∧
4.  if (sum != 0.0)                          ; Constraints for Payloads
5.    for (x = 0; x < n; x++)                ; (i will select a payload to test)
6.      v[x] /= sum;                         15. v_new[0] = payload[i]      ∧
7.  else                                     16. v_new[1] = payload[i+1]    ∧
8.    for (x = 0; x < n; x++)                17. v_new[2] = payload[i+2]    ∧
9.      v[x] = 1. / n;                       18. v_new[3] = payload[i+3]
```

Fig. 6. Controllable operations in 456.hmmer program.

Interestingly, we also observe that some operations of complex types and performing complex computations are determined as controllable by our analysis. Consider the following example that leverages existing floating point operations to construct malicious payloads. According to the IEEE-754 floating point representation standard, even a very small floating point value can affect all the 4 bytes of its presentation. For example, a floating point variable 0.0001 is encoded as 0x38d1b717 in memory. Fig. 6 shows `FNorm()` in 456.hmmer from SPEC. It first adds all elements in `v` into `sum` using `FSum()`, and then each element is divided by the `sum` if the `sum` is not 0.0. If the `sum` is 0.0, all the elements in `v` have `1.0 / n` where `n` is the size of `v`. Note that when there are multiple definitions of a variable (e.g., `v[x]`), A2C disjoins the constraints for these definitions, which are represented in the SSA form. The solver returns SAT for the constraints. The exploit input is a sequence of values (e.g., $-12068, -18966, -14108, -13991, ...$) whose binary representations do not denote any meaningful payload. But they are transformed to a meaningful payload by the operations in Fig. 6. The payload issues a system call through `int 0x80` with arguments.

B. Static Analysis to Compute Decoding and Encoding Sets

In this section, we discuss the second phase, i.e., the computation of decoding and encoding sets.

Language. A2C works on the Single Static Assignment (SSA) LLVM IR, which is generated from program source code. To facilitate precise discussion, we introduce a simplified language which models the LLVM IR in Fig. 7.

Memory loads and stores are denoted by `LOAD`(x_a) and `STORE`(x_a, x_v), respectively, with x_a holding the address and x_v the value. The address of a field access is explicitly computed by $x := x_{base} \rightarrow f$ with x_{base} the base pointer and f the field. Array accesses can be considered as a special kind of field accesses. `F`(x_a) models a call to function `F` with x_a the actual argument and x_f the formal argument. Function return is modeled by `ret`.

$$
\begin{array}{lll}
\textit{Program} & P & ::= s \\
\textit{Stmt} & s & ::= s_1;\ s_2 \mid \texttt{skip} \mid x :=^{\ell} e \mid x :=^{\ell} \texttt{LOAD}(r_a) \mid \\
 & & \texttt{STORE}^{\ell}(x_a, x_v) \mid \texttt{F}^{\ell}(x_a) \mid \texttt{ret}^{\ell} \mid \texttt{goto}^{\ell}(\ell) \mid \\
 & & \texttt{if}\ (x^{\ell})\ \texttt{then goto}(\ell_1) \mid \texttt{strcat}^{\ell}(x_{a1}, x_{a2}) \mid \\
 & & x := \texttt{lib}^{\ell}(x_1, x_2, ...) \mid x := \texttt{malloc}^{\ell}(x_s) \mid \\
 & & x := \phi^{\ell}(y, x_1, x_2) \mid \texttt{input}^{\ell}(x_{buf}, x_{size}) \\
\textit{Operator} & op & ::= + \mid - \mid * \mid / \mid < \mid > \mid == \mid ... \\
\textit{Expr} & e & ::= x \mid c \mid x\ \texttt{op}\ c \mid x_1\ \texttt{op}\ x_2 \mid x \rightarrow f \\
\textit{Var} & x & ::= \{x_1, x_2, x_3, ...\} \\
\textit{Const} & c & ::= \{true, false, 0, 1, 2, ...\} \\
\textit{Label} & \ell & ::= \{\ell_1, \ell_2, \ell_3, ...\}
\end{array}
$$

Fig. 7. Language.

Conditional or loop statements are not directly modeled. Instead we define jumps using `goto` and guarded `goto`. Conditional and loop statements can be constructed by combining jumps and guarded jumps. $\texttt{strcat}(x_{a1}, x_{a2})$ denotes a function that concatenates two strings. It appends the second string denoted by pointer x_{a2} to the first string x_{a1}. We define $\texttt{lib}(x_1, x_2, ...)$ to model library calls. It takes several x_n's as arguments and returns a value in another variable. Function $\texttt{input}(x_{buf}, x_{size})$ models library calls that read inputs such as `read()` and `recv()`. The $x := \phi(y, x_1, x_2)$ denotes the ϕ function in SSA that determines the value of a variable at the joint point of two branches. In particular, if y is true, $x := x_1$ otherwise $x := x_2$. We also explicitly model heap allocation through the `malloc()` function.

Operator denotes uncontrollable (computed by the previous phase) or comparative operations. Each statement is annotated with a label, which can be intuitively considered as the line number of the statement in the program.

$$
\begin{array}{lll}
\textit{Addr} & a & ::= \ell \mid x \mid a.f \\
\textit{PointsTo} & \sigma & ::= (\textit{Addr} \mid \textit{Var}) \times \textit{Context} \rightarrow \mathcal{P}(\textit{Addr}) \\
\textit{Source} & \textit{SRC} & ::= \texttt{CONST}(\ell,\ x) \mid \texttt{MARKED}(\ell,\ x) \\
\textit{TaintStore} & \tau & ::= (\textit{Addr} \mid \textit{Var}) \times \textit{Context} \rightarrow \mathcal{P}(\textit{Source}) \\
\textit{Context} & C & ::= \overline{\ell} \\
\textit{DecodeSet} & \textit{DEC} & ::= \mathcal{P}(< \textit{Context}, \textit{Label}, \textit{Var} >) \\
\textit{EncodeSet} & \textit{ENC} & ::= \mathcal{P}(< \textit{Label}, \textit{Var} \mid \textit{Const} >)
\end{array}
$$

$\texttt{ChkSrc}(\ell,\ x)\ ::=$
  $\texttt{if MARKED}(\ell_m,\ x_m)\ \in \tau^{\ell}(x, C)\ \texttt{then}$
    $\textit{DEC} := \textit{DEC}\ \cup\ \{< C, \ell, x >\}$
  $\texttt{if}\ (\{C, \ell, x\} \in \textit{DEC})\ \texttt{then}$
    $\texttt{foreach CONST}(\ell_c,\ c)\ \in \tau^{\ell}(x, C)\ \texttt{then}$
      $\textit{ENC} := \textit{ENC}\ \cup\ \{< \ell_c,\ c >\}$

$\texttt{ChkStrcat}(\ell,\ x_{a1},\ x_{a2})\ ::=$
  $\texttt{if}\ \exists a \in\ \sigma^{\ell}(x_{a1}, C),\ \texttt{MARKED}(\ell_m, x_m)\ \in \tau^{\ell}(a, C)\ \texttt{then}$
    $\texttt{if}\ \exists b \in\ \sigma^{\ell}(x_{a2}, C),\ \texttt{CONST}(\ell_c, c)\ \in \tau^{\ell}(b, C)\ \texttt{then}$
      $\textit{ENC} := \textit{ENC}\ \cup\ \{< \ell_c, c >\}$
  $\texttt{if}\ \exists a \in\ \sigma^{\ell}(x_{a2,C}),\ \texttt{MARKED}(\ell_m, x_m)\ \in \tau^{\ell}(a, C)\ \texttt{then}$
    $\texttt{if}\ \exists b \in\ \sigma^{\ell}(x_{a1}, C),\ \texttt{CONST}(\ell_c, c)\ \in \tau^{\ell}(b, C)\ \texttt{then}$
      $\textit{ENC} := \textit{ENC}\ \cup\ \{< \ell_c, c >\}$

$\texttt{TaintConst}(\ell,\ x,\ c)\ ::=$
  $\texttt{if}\ \{< \ell, c > \in \textit{ENC}\}\ \texttt{then}$
    $\tau^{\ell}(x, C) := \{\texttt{MARKED}(\ell, c)\}$
  $\texttt{else}$
    $\tau^{\ell}(x, C) := \{\texttt{CONST}(\ell, c)\}$

Fig. 8. Definitions for Abstract Interpretation Rules.

C. Static Analysis Phase

We formulate the static analysis as an abstract interpretation process. Intuitively, abstract interpretation can be considered as "*executing*" the program on the *abstract domain* instead of the concrete domain. The abstract domain is specific to an analysis. In abstract interpretation, it is often the case that branch outcomes cannot be statically determined. Therefore, it assumes all branches are possible. In the presence of loops, the interpretation may go through the loop bodies multiple times until a fix point is reached. If the abstract domain is well designed, the interpretation procedure is guaranteed to terminate.

Before the abstract interpretation, constants are propagated during preprocessing using an existing LLVM pass (e.g., $x_1 * x_2$ is rewritten to $x_1 * c$ if x_2 is determined to hold a constant c). During the analysis, A2C iteratively goes through program statements following the control flow and updating the corresponding abstract states (e.g., the decoding set) until a fix point is reached. Specifically, A2C taints input buffers from untrusted sources. The taints are propagated through controllable operations, which may be conducted through library functions (e.g., `memcpy()`, `toupper()`, and `icon v()`), linear operations (e.g., $y = x$ and $y = 3 * x$), and so on. If a tainted value reaches an operation on the decoding frontier computed in the previous phase, which includes comparative, uncontrollable, and terminal operations, taint propagation is terminated and the operation is added to the decoding set. However, the decoding set may be context-sensitive and path-sensitive. To handle such cases, statements that load constant values may need to be considered as sources and hence encoded. As a result, more statements may be added to the encoding set and the decoding set.

Definitions. To facilitate discussion, we introduce a few definitions in Fig. 8. Our analysis computes four kinds of abstract information: the points-to set, the taint set, and the encoding and decoding sets. The points-to set σ is a mapping from an abstract address a (representing some memory location) or a variable x, together with the calling context, to a set of abstract addresses denoting the memory locations that may be pointed-to by a or x. Abstract address *Addr* is denoted by some variable representing an abstract global/stack array/buffer or a label denoting an abstract heap buffer, followed by a sequence of fields. Intuitively, one can consider it as the reference path to some abstract memory location. The role of abstract addresses in our static analysis is similar to that of concrete addresses in dynamic analysis (e.g., to look up taint values). Since our analysis is context-sensitive and field-sensitive, context is part of the mapping and fields are explicitly modeled in abstract addresses.

Source represents the (taint) source of a value. There are two types of *Source*: `CONST` and `MARKED`, meaning a constant value and an untrusted input source, respectively. We use the term `MARKED` to indicate that a value originates from some input buffer and has only gone through controllable operations. Hence it is in the exploitable space (Section II). Such values shall be in their encoded form at runtime. We track the `MARKED` value propagation through our analysis. $\textit{TaintStore}$ τ stores the (taint) source information for abstract addresses and variables. Both σ and τ are flow-sensitive, meaning that A2C computes separate σ and τ for different program locations (i.e., labels). For example, we use τ^{ℓ} to denote the abstract taint mapping computed at ℓ. It is implicit in the rest of the paper for simplicity in discussion.

If `MARKED` values reach an operation on the decoding frontier, the operation is inserted to the $\textit{DecodeSet}$ $\textit{DEC}$. The $\textit{EncodeSet}$ $\textit{ENC}$ contains the set of statements at which the (input) values ought to be encoded. *Context* C is denoted

TABLE I. ABSTRACT INTERPRETATION RULES.

Statement	Interpretation Rule	Name
$\texttt{input}^{\ell}(x_b, x_s)$	foreach $a \in \sigma^{\ell}(x_b, C)$ $\tau^{\ell}(a, C) := \texttt{MARKED}(\ell, x_b)$; $ENC := ENC \cup \{\langle \ell, x_b \rangle\}$;	INPUT
$x :=^{\ell} x_1$ $(x =^{\ell} x_1 \text{ op } c)$	$\sigma^{\ell}(x, C) := \sigma^{\ell}(x_1, C)$; $\tau^{\ell}(x, C) := \tau^{\ell}(x_1, C)$;	NON-DF-OP
$x :=^{\ell} \texttt{LOAD}(x_a)$	$\sigma^{\ell}(x, C) := \bigcup_{\forall a \in \sigma^{\ell}(x_a, C)} \sigma^{\ell}(a, C)$ $\tau^{\ell}(x, C) := \bigcup_{\forall a \in \sigma^{\ell}(x_a, C)} \tau^{\ell}(a, C)$	LOAD
$\texttt{STORE}(x_a, x_v)$	$\forall a \in \sigma^{\ell}(x_a, C) : \sigma^{\ell}(a, C) \cup := \sigma^{\ell}(x_v, C)$ $\forall a \in \sigma^{\ell}(x_a, C) : \tau^{\ell}(a, C) \cup := \tau^{\ell}(x_v, C)$	STORE
$x :=^{\ell} x_1 \text{ op } x_2$	$\sigma^{\ell}(x, C) := \bot$; $\texttt{ChkSrc}(\ell, x_1)$; $\texttt{ChkSrc}(\ell, x_2)$;	DF-OP
$x :=^{\ell} x_1 \rightarrow f$	$\sigma^{\ell}(x, C) := \{a \cdot f \mid \forall a \in \sigma^{\ell}(x_1, C)\}$	FIELD
$x :=$ $\texttt{lib}^{\ell}(x_1, x_2, ..)$	for each $x_i \in \{x_1, x_2, ...\}$ $\texttt{ChkSrc}(\ell, x_i)$;	DF-TERM
$x :=^{\ell} c$	$\texttt{TaintConst}(\ell, x, c)$;	CONST
$\texttt{strcat}^{\ell}(x_{a1}, x_{a2})$	$\texttt{ChkStrCat}(\ell, x_{a1}, x_{a2})$;	STRCAT
$\texttt{F}^{\ell}(x_a)$	$C_0 := C$; $C := C \cdot \ell$; // x_f formal arg $\sigma^{\ell}(x_f, C) := \sigma^{\ell}(x_a, C_0)$; $\tau^{\ell}(x_f, C) := \tau^{\ell}(x_a, C_0)$; foreach buffer var $y \in F$: $\sigma^{\ell}(y, C) = \{y\}$;	CALL
ret	$C := C - \texttt{last}(C)$;	RET
$x := \phi^{\ell}(y, x_1, x_2)$	$\sigma^{\ell}(x, C) := \sigma^{\ell}(x_1, C) \cup \sigma^{\ell}(x_2, C)$; $\tau^{\ell}(x, C) := \tau^{\ell}(x_1, C) \cup \tau^{\ell}(x_2, C)$;	PHI
$x := \texttt{malloc}^{\ell}(x_s)$	$\sigma^{\ell}(x, C) := \ell$;	HEAP

by a sequence of labels (ℓ's) that models a call stack. Each element in the DEC set includes a *Context*, suggesting that we decode input buffers depending on the calling context. For example, $\langle C, \ell, x \rangle \in DEC$ suggests that when the statement denoted by ℓ is encountered under context C at runtime, A2C will decode the variable x.

conf.c

```
    VOID Read_Config(VOID){
      . . .
386:  fd = fopen( NGIRCd_ConfFile, "r" );
      . . .
441:  if( !fgets( str, ..., fd )) break;
442:  ngt_TrimStr( str );
```

parse.c

```
    Parse_Request(..., CHAR *Request){
      . . .      /* Request is a user request
                    through network. */
140:  ngt_TrimStr( Request );
```

tool/tool.c

```
    VOID
    ngt_TrimStr(CHAR *String) {
      . . .
      // String can be either from
      // a configuration file or
      // a network message
40:   start = String;
      . . .
46:   ptr = strchr( start, '\0' ) – 1;
47:   while(((*ptr == ' ') || (*ptr == 9) ||
          (*ptr == 10) || (*ptr == 13) || ...
      . . .
```

Fig. 9. An Example of Context Sensitive Code.

Decoding Set is Context-Sensitive and Path-Sensitive. The membership of a statement in the decoding set may change with the context. Fig. 9 shows an example in `ngircd`, an Internet Relay Chat (IRC) daemon program. In this example, we treat all network functions as untrusted input sources. Thus, the input data from these functions are encoded while data from files are not. `ngt_TrimStr()` is a utility function for trimming a string. It is invoked at different places. For instance, `Read_Config()` calls it with a string from the configuration file, which is not encoded. On the other hand, `Parse_Request()` also calls it, but with a string from the network. The string is encoded this time. Hence, A2C may or may not decode the value in `*ptr` at Line 47, depending on the context. Therefore, each statement in the DEC set is annotated with a context such that decoding is only performed when the same context is encountered at runtime.

The decoding set is also path-sensitive. Consider the example in Fig. 10 (a), which contains code snippets from `unrtf`, a program for converting documents in Rich Text Format (RTF) to other formats such as HTML and LaTeX. At ② and ③, `str` may hold a constant value or a tainted value `ch`. At ④ and ⑤, `str` is inserted to a hash map. Strings in the hash map are loaded and used at ⑥. Depending on whether ② or ③ is executed, Line 336 may or may not belong to the decoding set. In other words, if `tmp` holds a constant string at 336, it does not need to be decoded. Note that in this case, the context of Line 336 cannot be used to distinguish the different behaviors of the line. We cannot afford to track paths at runtime either. Hence, our solution is to identify the related constant strings, such as that at Line 326, and treat them as input sources so that they will be encoded as well. As a result, the behavior at Line 336 becomes path insensitive, always requiring decoding. □

Abstract Interpretation Rules. The interpretation procedure is formulated by the rules in Table I, which specify how the abstract information is updated upon each statement. Specifically, when the program reads data from untrusted input sources through $\texttt{input}(x_b, x_s)$ with x_b the buffer address and x_s the size, the *TaintStore* of all the abstract memory locations pointed to by x_b are set to `MARKED` (Rule INPUT). Note that using the context C makes our analysis context sensitive. The encoding set is also updated. Rule NON-DF-OP describes the interpretation of an operation that is not on the decoding frontier, i.e., controllable operation such as copy. In this case, A2C copies the points-to set and the abstract taint set. Rule LOAD describes that for a load instruction, the resulting points-to/taint set is the union of all the points-to/taint sets of all abstract memory locations pointed-to by the address x_a. Similarly, for a store statement, the points-to/taint set of the value variable x_v is added to the points-to/taint set of any abstract memory location pointed to by x_a. A2C only propagates taints for controllable operations. Rules DF-OP handles an uncontrollable operation or a comparative operation. It first resets the taint. It then calls function $\texttt{ChkSrc}(\ell, x)$ that checks if variable x is tainted as `MARKED`. If so, the statement together with the current context and the variable are inserted to the decoding set DEC. The context and variable information is needed to indicate which variable should be decoded and under what context. The function further tests if the statement is already in DEC and the variable is currently tainted as `CONST`, suggesting that the statement sometimes uses a value from untrusted input and sometimes uses a constant. This corresponds to the case in which the decoding set is path sensitive. To eliminate such path sensitivity, A2C adds the source of the constant to ENC, indicating that the source should be tainted as `MARKED` in the next round of abstraction interpretation.

Rule DF-TERM handles the other kind of operations in the decoding frontier: the terminal operations.

Rule CONST handles constant assignment, including constant string assignment. It tests if the constant assignment has been inserted to the ENC set (by Rules DF-OP or DF-TERM), indicating that the constant should be encoded so that we need

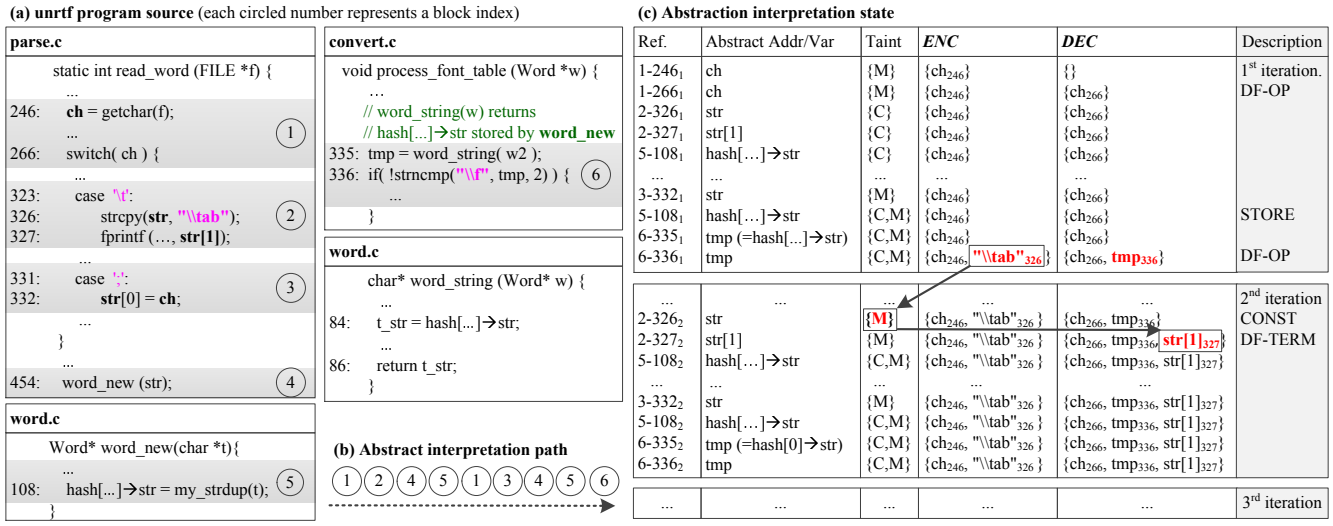

Fig. 10. An example of the iterative interpretation procedure on `unrtf`.

to figure out its decoding places. In this case, it sets the taint as `MARKED`, otherwise `CONST`. Rule STRCAT handles string concatenations. When a string from an untrusted source is concatenated with a constant string, we add the constant string to the ENC set to indicate that the string shall be encoded. Such concatenation happens frequently when a program uses string formatting functions such as `sprintf()`. Rule CALL updates the current context. It further propagates the points-to and taint sets from the actual argument to the formal argument. At the end, it sets the points-to sets of all the local buffer variables to contain themselves. The RET rule pops the last entry in the context. The PHI rule specifies that since x takes the value of either x_1 or x_2, its abstract sets are the union of those of x_1 and x_2. A2C does not model path conditions so that it essentially considers all paths are feasible and computes merged results along various paths. Rule HEAP describes that we use the label of the allocation statement to denote the abstract heap region allocated. In addition, the σ and τ entries computed at a location are also propagated to its control flow successors. The rules are omitted as they are standard. The abstract interpretation is iterative until a fix point is reached. It is easy to infer that our analysis must terminate as all the abstract domains are finite.

Example. Fig. 10 shows how the analysis works for `unrtf` that reads an RTF file and transforms it to various formats. Fig. 10 (a) shows some code snippets of the program. The description of them can be found at the beginning of Section IV-C. The program is simplified and slightly changed from its original version for illustration.

The abstract interpretation procedure is equivalent to traversing the path in Fig. 10 (b). The real interpretation order inside A2C is slightly different due to the ϕ functions that are omitted for easy explanation, although the outcome is identical. In the path, the two branches of the `switch` are traversed in two sub-paths: ①②④⑤ and ①③④⑤. They insert strings to the hash table and the strings are later accessed at ⑥.

Fig. 10 (c) shows the abstract states computed by A2C in multiple rounds. Each round follows the path in (b) during interpretation and corresponds to a sub-table in (c). The first column shows the block, line and round numbers of each statement. For instance, 2-326$_1$ means Line 326 inside ② in the first round of interpretation. Here, we only show the statements related to our analysis. The next two columns present the abstract address or variable that each statement accesses and its taint set. C means the `CONST` type and M denotes the `MARKED` type. The next two columns show the contents of ENC and DEC. The last column presents the rules applied.

First Round. ENC and DEC sets are empty at the beginning. At $1-246_1$, since `ch` is loaded from an input source, we add $\texttt{ch}_{246}$ to ENC to indicate that we should encode `ch` at Line 246. Then, `ch` is used in a comparison at $1-266_1$, thus we add $\texttt{ch}_{266}$ to DEC, meaning that we should decode `ch` at Line 266. For simplicity, we ignore the contexts in the DEC set. At $2-326_1$, a constant string is copied to `str`, and part of it is printed at $2-327_1$. Since `str` has a constant taint at this point, it does not need to be decoded. Later it is stored into the hash table at $5-108_1$. Then, a character from a file is copied to `str` at $3-332_1$, and is then stored in the hash table at $5-108_1$. Since A2C cannot distinguish if the hash table write and the previous write access different (abstract) memory locations, it unions the two taints so that the hash table is tainted with both `CONST` and `MARKED`, according to Rule STORE.

Later, at $6-335_1$ and $6-336_1$, the stored string is loaded and compared with a constant string "\\f". According to Rule DF-OP, since Line 336 is comparative and `tmp` is tainted with `MARKED`, it shall be decoded. An entry is hence inserted to the DEC set. Also according to the second `if` statement inside `ChkSrc()`, which is invoked by Rule DF-OP, the constant string at Line 326 is added to ENC, meaning that the constant string shall be encoded.

Second and Third Rounds. The second round traverses the same path. At $2-326_2$, the constant string is `MARKED` as it is in ENC, meaning that we should track its propagation to figure out the decoding places (Rule CONST). As a result, `str[1]` at Line 327 is added to DEC according to Rule DF-TERM. The rest is similar to the first round. In the third round, none of the abstract sets are updated, a fix point is reached. The analysis terminates.

From the final ENC and DEC sets, we should encode

at Lines 246 and 326, and decode `ch`, `str[1]` and `tmp` at Lines 266, 327 and 336, respectively. □

D. Runtime

Supporting Context Sensitivity. Once the analysis phase is finished, we have the DEC and ENC sets. Since both DEC and ENC are context sensitive, meaning that decoding and encoding should be performed only under certain calling contexts, the instrumentation needs to compare at runtime if the current context matches with that in DEC/ENC in order to perform decoding/encoding.

A straightforward way to obtain the current context is to perform stack walking. However, it incurs significant overhead. Furthermore, the resulting contexts are verbose and difficult to compare. To address the problem, we adopt a precise calling context encoding algorithm [64]. The algorithm maintains an *id* which is a *unique* number for each context. Given a program and its call graph, the algorithm automatically determines a unique *id* for each context. It further instruments the program in such a way that the instrumentation (at call sites) guarantees to produce the corresponding *id* when a context is reached. The instrumentation only requires simple (and low-cost) additions and subtractions before and after a subset of call sites. Context comparison becomes simple id comparison. Since the encoding algorithm is not our contribution, details are elided.

Encoding Based on One-Time-Dictionary. Simple encodings such as subtract-by-one are easy for the adversary to reverse engineer. He/she can prepare the exploit accordingly so that the exploit inputs become the plain-text payloads after our encoding. To address the problem, we use one-time-cipher. In particular, A2C has a large number of pre-generated random one-to-one mappings that project a byte to another unique byte. Whenever the program reads inputs from an untrusted source, A2C selects a mapping to encode the buffer. Since the dictionary for each untrusted input buffer is different from others, knowing previous mappings (e.g., through memory disclosure) does not help in launching subsequent attacks. More discussion can be found in Section V. Another thing we want to point out is that A2C mutates every byte from an untrusted sources. As such, none of the instructions from the original payload can be properly executed.

Using different dictionaries for different buffers requires A2C to track the dictionaries for individual buffers so that decoding can be properly performed. This is achieved by adding runtime taint propagation logic for controllable operations in the exploitable space. For controllable operations that are not simple copies (e.g., $y = 3 * x$), A2C decodes the source operand(s), performs the operation, and encodes the resulting operand using the same mapping. Since the exploitable space is very small, the entailed runtime overhead is low (see Section VI).

V. Threat Model

A2C assumes the subject program is benign but the inputs may be malicious. The user specifies which part of the inputs cannot be trusted such as network inputs and/or local file reads. It trusts the kernel. It also trusts that the low level *output* libraries are free of vulnerabilities, as it decodes the buffer values before calling these libraries. If they cannot be trusted, we can mitigate the problem by postponing the decoding to before output syscalls, which requires instrumenting libraries. Note that we do not trust all library functions. For example, we do not decode inputs for functions that copy data such as `strcpy` and `memcpy`. In practice, such functions are commonly exploited by attackers whereas output library functions such as `write` and `send` are not.

A2C aims to protect against payload injection attacks. It cannot handle other attacks that do not inject payload. It also requires the payload injection go through explicit input channels, which is true for most attacks. A2C currently only supports C/C++ programs and hence cannot deal with payload injections for programs in other languages such as JavaScript, although the idea is general.

Attacks In the Post-exploitable Space. A2C leverages constraint solving and a large pool of payload test cases that models the distribution of valid payloads to determine the decoding frontier with strong probabilistic guarantees. However, it may still be possible to construct some payloads via the very limited controllability of those uncontrollable operations on the decoding frontier. We argue that such payloads will have very limited functionalities. Moreover, we only protect against payloads that are larger or equal to 16 bytes. While it may be possible to construct payloads smaller than that, we again argue that such payloads will have very limited functionalities. Note that if a primitive value of four bytes is related to input, the attacker could inject a four byte payload to that primitive if there existed one. Protecting against such small payloads is almost impossible and unnecessary. In practice, we have not seen any examples of these payloads.

Memory Disclosure. Memory disclosure vulnerabilities can reveal memory contents of a process. Attackers can access memory pages that contain the encoded values and thus reverse engineer dictionaries. For example, he/she can manipulate the input by providing a sequence of unique values and then search in the disclosed memory for regions that have a sequence of unique values of the same length. By contrasting the two, the dictionary can be revealed. However, since A2C uses different dictionaries for individual input buffers, disclosing previous dictionaries does not help in subsequent attacks. Since A2C uses a random dictionary each time, it is really difficult to guess the next dictionary even knowing the previous dictionaries (i.e., 1 out N with N the number of pre-generated dictionaries). We use $N = 10^6$ in this paper.

VI. Evaluation

A2C is implemented on LLVM [2]. We evaluate A2C on 18 different real world programs shown in Table II. All the experiments were done on a machine with Intel Core i7 3.4GHz, 8GB RAM, and 32-bit LinuxMint 17.

We searched `exploit-db.com` to choose target programs. We tried the listed programs with reported exploits and selected those which we could reproduce. We have 6 network programs, with two client programs: `prozilla` and `stftp`, and four server programs: `apache`, `nginx`, `yops`, and `ngircd`. We have 12 user applications. `mupdf` reads and displays pdf documents. `unrar` is a decompressor program. `mcrypt` encrypts and decrypts files. `gif2png` converts gif to png. `unrtf` converts RTF files to other formats such as

HTML. `mp3info` reads and modifies meta tags of MP3 files. `rarcrack` and `fcrackzip` recover passwords of compressed files (e.g., `zip` and `rar` files) using different strategies. `vfu` is a text-mode file manager. `chemtool` is a GUI program for drawing chemical structures. `Xerces-C` is an XML parser. Among these programs, we have two GUI programs that require user interactions: `mupdf`, and `chemtool`. `vfu` requires text-based user interactions.

The first two columns of Table II show the programs and their size in C source code lines (CLOC). The third and fourth columns present the number of entries in DEC and ENC computed by our analysis. They are essentially LLVM IR statements annotated with contexts. The fifth column shows the number of statements in DEC that behave differently depending on the context. One such statement has multiple entries in the DEC set (for different contexts). The sixth column represents the number of instrumented IR statements for calling context encoding. The last two columns show the time spent on computing the decoding frontier, and the static analysis for DEC/ENC set computation and instrumentation, respectively. The overhead of decoding frontier computation includes the running time of Z3 constraint solver. We use one minute as the timeout threshold. We also avoid testing identical payload sequences.

From the table, we have the following observations. A2C can handle large and complex programs such as `mupdf` and `apache`. The number of entries in ENC/DEC is small with respect to the program size. This supports our speculation that the exploitable space is small. The data in the fifth column also supports that context sensitivity is needed. Finally, the analysis overhead is acceptable. Some large programs take a few hours. However, we argue that this is one-time cost.

TABLE II. EVALUATION RESULTS FOR ANALYSIS.

Program	Size	$\|ENC\|$	$\|DEC\|$	CS[1]	CCE[2]	Analysis Time	
						DF Comp.[3]	SA[4]
mupdf	483K	598	2283	241	172	1h 5m	12m 11s
prozilla	54K	98	754	391	104	9m 49s	2m 43s
stftp	18K	42	144	42	37	6m 51s	1m 58s
yops	9,215	49	153	4	12	24s	13s
nginx	335K	151	1005	37	72	34m 14s	17m 22s
ngircd	119K	123	391	113	249	7m 39s	10m 1s
unrar	99K	36	239	44	164	17m 21s	7m 11s
mcrypt	36K	83	278	40	35	12m 41s	4m 20s
gif2png	16K	32	129	28	22	8m 19s	1m 38s
mp3info	17K	33	91	23	19	6m 9s	2m 17s
fcrackzip	48K	18	37	23	11	8m 17s	2m 58s
chemtool	176K	100	388	27	39	20m 35s	7m 41s
vfu	180K	64	129	49	318	12m 51s	8m 21s
unrtf	25K	31	220	291	178	14m 5s	2m 43s
rarcrack	1,364	7	19	39	9	0s	5s
make	124K	106	719	125	94	31m 14s	1h 40m
Xerces-C	415K	121	1137	102	213	1h 28m	6h 21m
apache	208K	364	1586	98	63	1h 56m	5h 41m

[1]# of Context Sensitive Statements.
[2]# of instrumentations for Calling Context Encoding.
[3]Decoding Frontier Computation Phase. [4]Static Analysis Phase

A. Performance

Performance for Programs with Vulnerabilities (i.e., those in Table II). To evaluate the runtime overhead of A2C, we run both the original program and the instrumented version 10 times and take the average. We use large inputs. For example, we use document files that are larger than 10MB to test file processing programs `unrtf`, `Xerces-C`, and `gif2png`. As such, the native executions usually last for more than a few seconds. For the programs that require user interactions, we force them to quit after they load, process, and render the inputs, and before they take any user interactions. We manually identify the locations in the source files that indicate such status (e.g., before calling a function to change the status bar to show the input is successfully loaded and rendered) and insert `exit()` to these locations. We then measure the overhead for these shortened executions. Note that, this usually leads to over-approximation of the overhead as our instrumentation largely lies in the initial input loading and parsing logic.

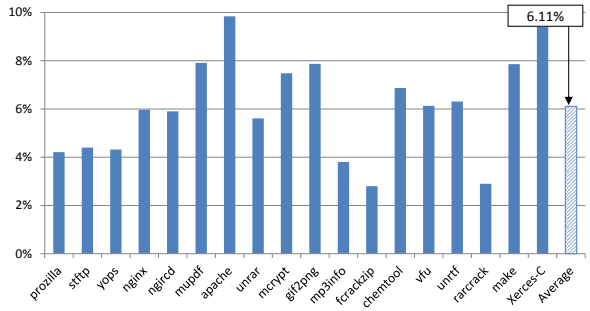

Fig. 11. Normalized Overhead on Programs in Table II.

Fig. 11 shows the result. The average overhead is 6.11%. In most cases, the overhead is less then 6%. There are a few exceptions. Programs dedicated to processing and parsing input files such as `make`, `Xerces-C`, `unrtf`, and `gif2png` have relatively higher overhead. This is because the instrumented statements are being executed throughout the execution. Also, the programs that require interactions, e.g., `mupdf`, `chemtool`, and `vfu`, have relatively higher overhead. This is because of the way we measure the overhead. `apache` has the highest overhead (9.84%) due to the complex structure of input filters that leads to many constant strings being encoded.

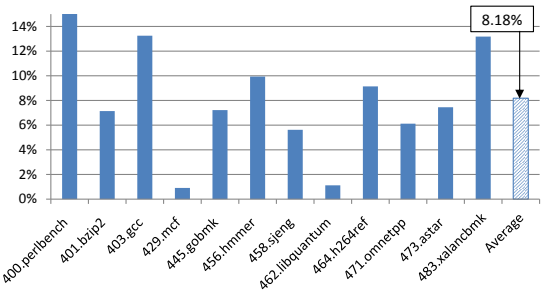

Fig. 12. Normalized Overhead on SPEC CPU2006.

SPEC CPU2006. We also evaluate the performance of A2C on SPEC CPU2006. We run both the original and instrumented programs 10 times using the reference inputs. Fig. 12 shows the result. The average overhead is 8.18%. `401.perlbench`, `403.gcc`, and `483.xalancbmk` have relatively higher overhead because they process inputs intensively. `456.hmmer` has 9.94% overhead as it processes inputs even during the execution of its main algorithm. `429.mcf` and `462.libquantum` have extremely low overhead, less than 1.5%. This is because they process inputs once at the very beginning. As such, A2C only needs to decode at the beginning and the rest of the execution does not cause any overhead. The average overhead for all 30 programs including programs in Table II and SPEC CPU2006 is 6.94% and the geometric mean is 5.94%.

TABLE III. EVALUATION RESULTS FOR ATTACK PREVENTION.

Program	# of Inputs (Mal./Benign)	# of Vulnerabilities	# of Payloads (Shellcode/ROP)	# of Crashes (Mal./Benign)	# of ins. exec. in Payloads	# of ROP Gadgets Exec. in Payloads	Precision/Recall
mupdf	10 / 20	1 (CVE-2014-2013)	50 / 50	1000 / 0	3.62	0.1	100% / 100%
mcrypt	10 / 20	2[1]	50 / 50	1000 / 0	3.62	0.18	100% / 100%
sftp	10 / 20	1 (EDB-ID: 9264)	50 / 50	1000 / 0	3.6	0.08	100% / 100%
yops	10 / 20	1 (EDB-ID: 14976)	50 / 50	1000 / 0	3.62	0.05	100% / 100%
nginx	10 / 20	1 (CVE-2013-2028)*	50 / 50	1000 / 0	3.62	0.09	100% / 100%
ngircd	10 / 20	2[2]	50 / 50	1000 / 0	3.62	0.11	100% / 100%
unrar	10 / 20	1 (EDB-ID: 17611)	50 / 50	1000 / 0	3.62	0.18	100% / 100%
prozilla	10 / 20	2[3]	50 / 50	1000 / 0	3.6	0.09	100% / 100%
gif2png	10 / 20	1 (CVE-2009-5018)	50 / 50	1000 / 0	3.62	0.09	100% / 100%
mp3info	10 / 20	1 (CVE-2006-2465)	50 / 50	1000 / 0	3.62	0.05	100% / 100%
fcrackzip	10 / 20	1 (EDB-ID: 14904)	50 / 50	1000 / 0	3.62	0.05	100% / 100%
chemtool	10 / 20	1 (EDB-ID: 36024)	50 / 50	1000 / 0	3.6	0.18	100% / 100%
vfu	10 / 20	1 (EDB-ID: 35450)	50 / 50	1000 / 0	3.61	0.18	100% / 100%
unrtf	10 / 20	1 (CVE-2004-1297)	50 / 50	1000 / 0	3.62	0.18	100% / 100%
rarcrack	10 / 20	2[4]	50 / 50	1000 / 0	3.62	0.05	100% / 100%
make	10 / 20	1 (EDB-ID: 34164)	50 / 50	1000 / 0	3.62	0.18	100% / 100%
Xerces-C	10 / 20	1 (CVE-2015-0252)	50 / 50	1000 / 0	3.62	0.07	100% / 100%
apache#	10 / 20	2[5]	50 / 50	1000 / 0	3.6	0.13	100% / 100%

[1](CVE: 2012-4409, 2012-4527) [2](CVE: 2005-0226, 2005-0199) [3](CVE: 2005-0523, 2004-1120)
[4](EDB-ID: 15062, 15054) [5](CVE: 2004-0940, 2006-3747) *This CVE includes multiple vulnerabilities #Version 1.3.31

B. *Effectiveness*

To evaluate the effectiveness of A2C in preventing attacks and allowing benign executions, for each program, we prepare 10 exploits and 20 other benign inputs. For each exploit input, we prepare 100 different malicious payloads, including 50 shellcodes and 50 ROP payloads.

The shellcodes are generated from [51], and we use ROP attack creators [52], [53] to generate 50 different ROP payloads for each vulnerable application. Thus, we have 1,000 attack executions and 20 benign executions for each program. Note that, as shown in Table III Column 3, some programs have more than one vulnerability, which require unique exploit inputs. The table also shows the results. Observe in the fifth column, A2C successfully crashes all the attacks and allows all the benign inputs to proceed to normal termination and produce the expected outcomes. The next two columns show the average number of payload/gadget instructions that got executed before crashing. They are all in very small numbers. As such, they can hardly cause any damage to the system.

Decoding Frontier (DF) Operation Classification. We further analyze the DF operations for all the subject programs and classify them into a few categories. Fig. 13 shows the results, from which we have the following observations.

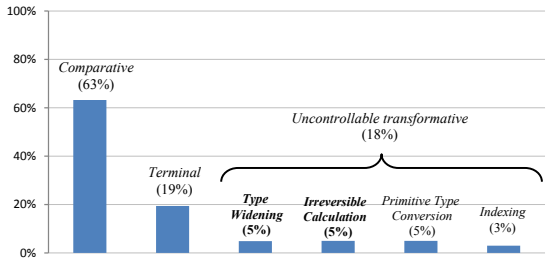

Fig. 13. Different Types of Decoding Frontiers.

First, 63% operations on DFs are *Comparative Operations*. Note that comparative operations are mostly conducted on individual buffer elements (of primitive types), A2C only decodes the element needed by the operation. The decoded value is dead (e.g., overwritten) right after the operation. Such DF operations cannot be exploited. Second, 19% DF operations are *Terminal Operations*. For a terminal operation, A2C first copies the original buffer to a temporary buffer, and then decodes the temporary buffer. Also, after the terminal operation, A2C releases the temporary buffer to minimize the attack window. Third, we also identify a few kinds of *Uncontrollable Transformative Operations*. In particular, *Type Widening* expands each element in a buffer by padding it with some specific byte(s) such as `0x00`. Note that we use the constraint solver to determine whether each case of type widening is controllable as not all type widening cases are uncontrollable. In fact, casting a one-byte data type to a two-byte data type is solvable in many cases. Note that some binary operations (e.g., multiplication) of values with smaller types yield a value of a large type. These are not type-widening as the bits in the resulting value are often fully/largely controllable. *Irreversible Calculation* means arithmetic transformations that cause intensive correlations among values so that the solver returns UNSAT for all tests. An example can be found in Section IV-A. *Primitive type conversion* means that a buffer element is converted to a value of primitive type (e.g., `atof()`) and this value is not stored to any array/buffer. Since single primitive values can hardly be exploited to inject payloads due to the size, decoding is safe. Note that A2C protects consecutive primitive values if they can form a region larger than 16 bytes. *Indexing* means that an encoded value is used to index a non-constant array. It is safe to decode the value because the decoded value is of a primitive type and soon dies after the operation. The entire buffer is never decoded.

Decoding Frontier (DF) Computation. Table. IV shows the evaluation results of decoding frontier computation. The first column shows the programs. The next three columns show the numbers of controllable operations, uncontrollable operations, and their sum, respectively. The last column shows the average number of constraints for each memory region under test. Recall that if the solver returns SAT, TIMEOUT or UNKNOWN for a constraint in any payload sequence test, the corresponding operations are considered controllable.

We make the following observations. First, in most cases, there are more UNSAT cases than SAT cases. This means that most input related computations are not controllable. There are a few exceptions. `gif2png`, `apache`, and `chemtool` have more SAT cases as our modeling of the external library

TABLE IV. RESULTS FOR DECODING FRONTIER COMPUTATION.

Program	# of Operations			Avg. # of Constraints
	Controllable	Uncontrollable	Total	
mupdf	9	141	150	16.4
Prozilla	4	20	24	15.9
stftp	2	8	10	11.5
yops	0	1	1	8
nginx	4	41	45	17.2
ngircd	2	12	14	14.1
unrar	6	33	39	14.2
mcrypt	4	24	28	18.3
gif2png	13	10	23	16.9
mp3info	4	9	13	15.3
fcrackzip	4	4	8	13.6
chemtool	29	22	51	14.1
vfu	3	25	28	15.5
unrtf	2	22	24	14.5
rarcrack	0	0	0	0
make	9	53	62	15.4
Xerces-C	14	75	89	14.8
apache	145	129	274	17.7
Average	**14.1**	**34.9**	**49**	**14.05**

calls is not complete and the modeling of floating point functions is conservative. For example, we assume `exp()` function can return any positive floating point values while the parameter of the `exp()` function may have constraints, hence it may not be able to produce some floating point values. Note that such a conservative assumption only causes over-approximation. Second, the total number of operations for testing is not large (`apache` has the largest number 274). This is because the controllability classification for most operations is straightforward (e.g., comparative operations and copy operations) and hence does not require constraint solving. Third, the average number of constraints in our tests is not large, suggesting that controllable operations are often shallow in the data flow, meaning that they are close to program inputs. This supports our assumption that most computation happens in the post-exploitable space. Note that we do not need to test controllability of operations if their operands are not controllable.

C. Case Studies

Running Web Servers on Real-world Traffic. To further evaluate the robustness of A2C, we run the instrumented web servers on a real-world traffic log. We obtained our institution's server access log from November 2015 to January 2016. The log contains 5.6 million requests with 4.2 million unique requests, including some suspicious requests with binary payloads (about 100 of them). We also randomly inject 300 exploit inputs to the access log. We ran three servers (`apache`, `nginx`, and `yops`) with these requests. The results show that the instrumented versions produce the same expected results as the original versions except for the attacks. All attacks are prevented. The throughput is only reduced by 8.83%, 7.37%, and 5.49%, respectively.

Code Injection Through Benign Functions and Payload Triggered Through Integer Overflow. In this case study, we show how a payload can be injected through benign and non-vulnerable program logic and later triggered by an integer overflow vulnerability. Such a combination makes it difficult for traditional defense techniques. Fig. 14 shows code snippets of the victim program, `mupdf`. First, observe that the `xps_read_dir_part()` function reads a file. It opens a file at Line 455, then gets the size of file at Line 458. Later, it reads the file and puts it into a heap buffer (`part->data`) at Line 462. Note that the function `xps_read_dir_part()` is not vulnerable. But still, the attacker can provide a crafted `xps` file that contains a malicious payload. The payload will be injected through the normal file read in the benign function. Thus, most existing protection schemes including CFI, DFI, ASLR, and boundary checkers cannot prevent such injection. While malicious payload detection methods can identify the injected shellcode by scanning the input file at the `fread` function, the attacker can use obfuscation techniques to circumvent such detection.

To trigger the payload, the attacker exploits an integer overflow vulnerability. The integer overflow happens as follows. It reads input from a file at Line 91 in `lex_number()`. Then the input is propagated to Line 97 where the integer overflow occurs. The program assumes the input `c` is between '0' to '9', and converts it into an index (`i`). At Line 106, the converted index is stored into `buf->i`. Later, the index is used to write elements into a structure (at Lines 176-178 in `pdf_repair_obj_stm()`). Note that the earlier index is propagated to variable `n` which is also used as an index. This integer overflow can be leveraged to overwrite some critical data fields such as function pointers in order to change control flow of the program to the injected shellcode. Note that the exploit may not be detected by address sanitizers as the attacker can manipulate the offset `n` to directly overwrite the target memory addresses that may fall into other legitimate memory regions, without overwriting the canaries.

In contrast, A2C defeats the attack by breaking its weakest link, which is the injected payload itself. In particular, A2C mutates the input including the shellcode at the `fread` in Line 462. The original shellcode is shown in Fig. 14 (a), and the corresponding mutated shellcode in Fig. 14 (b). Observe that the mutated shellcode is broken and not executable.

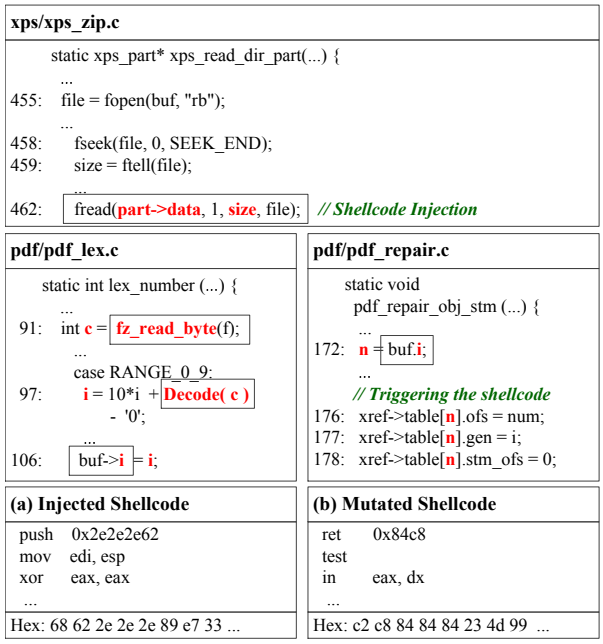

Fig. 14. Integer Overflow in `mupdf`.

Note that A2C does not prevent the integer overflow. Even through it encodes the input value at Line 91, it decodes the value right before the overflow (at Line 97) because that is an operation of primitive type. In other words, the attacker can

still exploit integer overflow vulnerabilities. However, when the control flow of the program is redirected to the injected shellcode, the execution crashes almost immediately as the first instruction of the mutated shellcode is "`ret 0x84c8`", which does not have a valid return address.

One might think the attacker can exploit the integer overflow to direct the control flow to some buffer in the post-exploitable space. However, as we pointed out in Section V, the transformations performed by the subject programs are complex enough that the attackers cannot generate plain-text payloads in the post-exploitable space.

Preventing ROP attacks. As DEP (Data Execution Prevention) becomes more and more popular, attackers now use ROP to bypass such protection. In this case study, we show how A2C prevents ROP attacks using an example.

convert.c

```
    void process_font_table (...) {
      ...
331:  char name[255];
      ...
341:  while (w2) {
342:    tmp = word_string(w2);
343:    if ( tmp &&
            Decode( tmp[0] ) != '\\' )
344:      strcat( name, tmp );
```

(a) Injected ROP gadgets

Address	Instructions
0x804d820	mov ebx,0x0; ret
0x804ec7d	mov eax,0x806275c; ret
...	...

(b) Mutated ROP gadgets

Address	Instructions
0xa2ae728a	Invalid address
0xa2ae46d7	Invalid address
...	...

Fig. 15. Stack Buffer Overflow in `unrtf`.

Fig. 15 shows `unrtf` which has a stack buffer overflow vulnerability. It can be leveraged to inject a malicious payload that allows constructing a ROP gadget chain. The program first gets a user provided string at Line 342. Then, it compares the string with a constant at Line 343. As it is a comparative operation, A2C decodes the value, allowing proper comparison. The buffer overflow happens when the program copies the user provided buffer (`tmp`) to a local buffer `name` at Line 344 in `process_font_table()`. Observe that the size of `name` is only 255. Thus, providing a long enough input to the `tmp` buffer will result in a stack overflow.

Fig. 15 (a) shows the injected ROP payload and the corresponding gadgets. The address column shows the payload that contains the raw addresses of the ROP gadgets. The instructions column shows the instructions from the ROP gadgets. Observe that they all end with a `ret` instruction. These chains of instructions are essentially the ones that get executed once the attack is launched. Fig. 15 (b) shows the mutated payload. For demonstration purpose, we use a simple encoding/decoding scheme even though our implementation uses one-time-dictionary. In particular, the mutation is to `xor` a value with `0xAA`. Observe that all the addresses in the original payload are encoded and point to invalid addresses. Hence, the attack fails. Note that since A2C prevents attacks by mutating payloads, the injection methods do not affect our protection.

Preventing English Shellcode. As a counter attack to shellcode detection techniques, Mason et al. proposed an automatic way to generate shellcode which is similar to English prose [39]. Such technique can be used to avoid existing shellcode identification techniques [67], [37], [45], [18].

Fig. 16 shows an example of English Shellcode presented in [39]. As shown in the ASCII column, the shellcode is an English statement. The corresponding assembly instructions are listed in the first column. While we are just showing

English Shellcode and Mutated English Shellcode

Assembly	Opcode	ASCII
push esp push 0x20657265 ...	54 68 65 72 65 20 ...	There is a majorcenter of economic activity, ...
inc dl **iret** ...	fe c2 cf ...	No ASCII character found

Fig. 16. English Shellcode Example.

one example, in practice attackers also use other various shellcode obfuscation and compression techniques [38], [59] to avoid shellcode identification. A2C mutates all untrusted inputs including shellcodes as they are part of the inputs. The mutated English Shellcode includes those shaded in Fig. 16. For demonstration, we again apply the `xor` with `0xAA` mutation. Observe that the mutated shellcode is completely different from the original shellcode. While the first instruction is executable, it does not help attackers to achieve anything useful. More importantly, the second instruction is `iret`, which can only be executed in a kernel mode. Executing `iret` results in a segmentation fault. One interesting observation is that the first a few instructions in the mutated shellcode are often executable. The fifth column of Table III shows the average number of instructions executed in the mutated payload is very small (<4). It is also important to note that such a few (mutated) instructions do not have the same semantics as the original malicious logic. They often immediately lead to crashes and do not cause any damage to the system.

Buffer Overflow In Structure. AddressSanitizer [56] is an important technique to prevent various buffer overflow attacks including heap and stack overflows. It works by placing canaries before and after a buffer. One of the limitations of the technique is that it cannot handle buffer overruns within a structure.

Program.c

```
    void process(RECORD* p) {
1:    fread( p->name, ... );
2:    printf("Name: %s\n",
            Decode( p->name ));
3:    p->handler( p->privilege );
```

Program.h

```
typedef struct tag_RECORD {
  char name[255];
  void (*handler)(int);
  int privilege;
} RECORD;
```

Fig. 17. Buffer Overrun in Structure.

Fig. 17 shows a buffer overflow vulnerability in a structure. Specifically, buffer `name` in the structure `RECORD` can affect adjacent data fields including a function pointer `handler`. At Line 1, it reads a file to fill the `name` buffer. By providing an input string longer than 255 bytes, it can overwrite `handler`. Note that A2C mutates the input in `fread` at Line 1, the `handler` is overwritten with a mutated address. Then, the program calls `printf` to display the name on the screen. As `printf` is an external call, A2C decodes the input buffer `name`. Specifically, in our implementation of the decoding function, when A2C decodes a buffer for a library call, it allocates a new buffer, copies the original encoded buffer, and then decodes it in the new buffer before passing it. Since A2C does not decode the original buffer, the injected malicious payload remains mutated. At Line 3, the program calls `handler`. Although it is overwritten, the function pointer no longer points to the injected shellcode. Note that the `privilege` field can also be overwritten to launch non-control data attacks [16]. A2C mitigates the attacks by encoding the inputs from untrusted sources. As a result, the attacker cannot control the overwritten value.

VII. Related Work

Control-flow Integrity (CFI). Recent advances in control-flow integrity have developed very robust systems for preventing malicious/abnormal control flows within a victim program. These typically monitor execution to enforce pre-determined control flow paths [44], [9], [66], [31], [73], [74], [72], [43], [68], [40]. In contrast, A2C provides protection by corrupting input payloads, which is a perspective orthogonal to the enforcement of a program's legitimate control flow graph. Therefore, A2C is complementary to and can be deployed alongside CFI, e.g., to prevent exploit injection attacks that may employ indirect calls or not violate control flow integrity [24], [29], [15], [55], [54], [40], [19], [14].

Malicious Payloads Detection. In [67] and [37], researchers proposed analyzing inputs to detect malicious payloads with little runtime overhead. However, Fogla at el. [28] demonstrated that polymorphism techniques can defeat these approaches. Dynamic analysis using emulation [46], [61] have been proposed to uncover polymorphic payload injection attacks, but they cause non-negligible performance penalty. A2C mutates all input buffers from untrusted sources and thus is resilient to polymorphism. It does not require emulation and causes low overhead. Nozzle [48] proposed a novel technique to detect heap spraying attacks at runtime. It uses runtime interpretation and static analysis to analyze suspicious objects in the heap. While Nozzle focuses on detecting heap spraying on JavaScript, A2C takes a more general approach to prevent a wider range of input injection attacks.

Randomization Approaches. Address space layout randomization (ASLR) is one of the most widely deployed defense mechanism to mitigate payload injection and triggering. ASLR randomizes the memory layout of a program when the OS loads the binary and dynamic libraries. ASLR is already a default defense mechanism in most operating systems including Linux, MacOS, BSD, and Windows. Address space layout perturbation [34] and fine-grained randomization techniques [70], [42], [7], [22], [17], [30] have been developed to provide higher entropy. Instruction set randomization [33], [47], [41] aims to change the underlying instruction set to prevent executing injected code. However, it was shown recently that randomization could be evaded by brute-force attacks [58], [8], memory disclosure attacks [11], [57], [36], and just-in-time code reuse attacks [62]. In [23], researchers presented a novel defense technique to mitigate counterfeit object-oriented programming (COOP) attacks [54]. They randomize the layout of the code pointer table and plant booby-traps to prevent brute-force attacks. Compared to these techniques, A2C provides protection by working from the input perspective, which is complementary to randomization. Data randomization [6], [13] dynamically decrypts a buffer upon each buffer access and encrypts it again after the access. It encrypts all buffers including those not related to inputs. It also uses different keys for various buffers. A2C shares a similar idea of buffer encoding with data randomization. The differences lie in that A2C focuses on input related buffers; it encodes only once for each input and decodes only at the decoding frontier. As such, A2C has relatively lower overhead. PointGuard [20] encrypts pointer values at runtime.

Bounds Checking. Stackguard [21] inserts a secret value (canary) before each return address and frame pointer. However, it can be defeated through information leak attacks that reveal a canary value [49], [12]. Compile-time code analysis [69], [35] have been proposed to detect unsafe array and pointer accesses. However, they often generate many false positives and focus on specific kinds of vulnerabilities. Cling [4] and AddressSanitizer [56] provide pointer safety to prevent exploiting pointer related bugs such as use-after-free. However, as shown in our case study, they can hardly handle advanced attacks [71]. In contrast, A2C aims to break the weakest link of attacks, which is the payload itself.

VIII. Conclusion

We present A2C that provides general protection against a wide spectrum of payload injection attacks. It mutates all input buffers from untrusted sources to break malicious payloads. To assure the program functions correctly on legitimate inputs, it decodes them right before they are used to produce new values. A2C automatically identifies such places at which it needs to decode using a novel constraint solving based approach and a sophisticated static analysis. Our experiments on a set of real-world programs show that A2C effectively prevents known payload injection attacks on these programs with reasonably low overhead (6.94%).

Acknowledgment

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668 and 0845870, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] Exploits database by offensive security. https://www.exploit-db.com/.

[2] The llvm compiler infrastructure. http://llvm.org/.

[3] Penetration testing software. metasploit. https://www.metasploit.com/.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *S&P'08*.

[5] C. Anley. Creating arbitrary shellcode in unicode expanded strings, the "venetian" exploit. https://www.helpnetsecurity.com/dl/articles/unicodebo.pdf, 2002.

[6] S. Bhatkar and R. Sekar. Data space randomization. In *DIMVA'08*.

[7] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *CCS'15*.

[8] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *S&P'14*.

[9] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *ACSAC'11*.

[10] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *ASIACCS'11*.

[11] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *CCS'08*.

[12] Bulba and Kil3r. Bypassing stackguard and stackshield. http://phrack.org/issues/56/5.html, 2000.

[13] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120.

[14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *SEC'15*.

[15] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *SEC'14*.

[16] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *SEC'05*.

[17] Y. Chen, Z. Wang, D. Whalley, and L. Lu. Remix: On-demand live randomization. In *CODASPY'16*.

[18] R. Chinchani and E. van den Berg. A fast static analysis approach to detect exploit code inside network flows. In *RAID'05*.

[19] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *CCS'15*.

[20] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities. In *SEC'03*.

[21] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *SEC'98*.

[22] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *S&P'15*.

[23] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *CCS'15*.

[24] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *SEC'14*.

[25] L. De Moura and N. Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, Berlin, Heidelberg. Springer-Verlag.

[26] T. DETRISTAN, T. ULENSPIEGEL, Y. MALCOM, and V. UNDERDUK. Polymorphic shellcode engine using spectrum analysis. http://phrack.org/issues/61/9.html, 2003.

[27] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap taichi: Exploiting memory allocation granularity in heap-spraying attacks. In *ACSAC'10*.

[28] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *SEC'06*.

[29] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *SEC'14*.

[30] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the S&P'12*.

[31] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *NDSS'14*.

[32] K2. Admmutate documentation. http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz, 2003.

[33] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS'03*.

[34] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC'06*.

[35] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *SEC'01*.

[36] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From zygote to morula: Fortifying weakened aslr on android. In *S&P'14*.

[37] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *CCS'05*.

[38] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS'03*.

[39] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *CCS'09*.

[40] B. Niu and G. Tan. Per-input control-flow integrity. In *CCS'15*.

[41] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. Asist: architectural support for instruction set randomization. In *CCS'13*.

[42] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programm ing using in-place code randomization. In *S&P'12*.

[43] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *SEC'13*.

[44] P. Philippaerts, Y. Younan, S. Muylle, F. Piessens, S. Lachmund, and T. Walter. Code pointer masking: Hardening applications against code injection attacks. In *DIMVA'11*.

[45] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *ACSAC'10*.

[46] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *RAID'07*.

[47] G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *ACSAC'10*.

[48] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *SEC'09*.

[49] G. Richarte et al. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[50] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[51] J. Salwan. Shellcodes database for study cases. http://shell-storm.org/shellcode/.

[52] S. Schirra. ROPgadget - Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/.

[53] S. Schirra. Ropper - rop gadget finder and binary information tool. https://scoding.de/ropper/.

[54] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *S&P'15*.

[55] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-rop defenses. In *RAID'14*.

[56] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-sanitizer: A fast address sanity checker. In *ATC'12*.

[57] F. Serna. Cve-2012-0769, the case of the perfect info leak. http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[58] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS'04*.

[59] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *NDSS'08*.

[60] Skylined. http://www.edup.tudelft.nl/b̃jwever/advisory_iframe.html.php.

[61] K. Snow, S. Krishnan, F. Monrose, and N. Provos. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *SEC'11*.

[62] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *S&P'13*.

[63] M. Suenaga. Evolving shell code. Whitepaper, Symantec Security Response, Japan, 2006.

[64] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *ICSE'10*.

[65] M. D. Team. Metasploit project. http://metasploit.com, 2006.

[66] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *SEC'14*.

[67] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *RAID'02*.

[68] 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'15*.

[69] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS'00*.

[70] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *CCS'12*.

[71] E. Wimberley. Bypassing AddressSanitizer. https://packetstormsecurity.com/files/123977/Bypassing-AddressSanitizer.html.

[72] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *DSN'12*.

[73] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *S&P'13*.

[74] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *SEC'13*.