Block Oriented Programming: Automating Data-Only Attacks

Kyriakos K. Ispoglou
ispo@purdue.edu
Purdue University

Bader AlBassam
balbassa@purdue.edu
Purdue University

Trent Jaeger
tjaeger@cse.psu.edu
Pennsylvania State University

Mathias Payer
mathias.payer@nebelwelt.net
Purdue University

ABSTRACT

With the wide deployment of Control-Flow Integrity (CFI), control-flow hijacking attacks, and consequently code reuse attacks, are significantly harder. CFI limits control flow to well-known locations, severely restricting arbitrary code execution. Assessing the *remaining attack surface* of an application under advanced control-flow hijack defenses such as CFI and shadow stacks remains an open problem.

We introduce BOPC, a mechanism to assess whether an attacker can execute arbitrary code on a CFI/shadow stack hardened binary automatically. BOPC leverages SPL, a Turing-complete high-level language that abstracts away architecture and program-specific details, such as register mappings, to express exploit payloads. SPL payloads are compiled into a program trace that executes the desired behavior on top of the target binary. The input for BOPC is an SPL payload, a starting point (e.g., from a fuzzer crash), and an arbitrary read/write primitive that allows application state corruption. To map SPL payloads to a program trace, BOPC introduces *Block Oriented Programming* (BOP), a new code reuse technique that utilizes entire basic blocks as gadgets along valid execution paths in the program, i.e., without violating CFI policies. We find that the problem of mapping payloads to program traces is NP-hard, so BOPC first reduces the search space by pruning infeasible paths and then uses heuristics to guide the search to probable paths. BOPC encodes the BOP payload as a set of memory writes.

We execute 13 SPL payloads applied to 10 popular applications. BOPC successfully finds payloads and complex execution traces – which would likely not have been found through manual analysis – while following the target's Control-Flow Graph under an strict CFI policy in 81% of the cases.

ACM Reference format:
Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of Technical Report, West Lafayette, USA, 9 May 2018,* 16 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Control-flow hijacking and code reuse has been a challenging problem for applications written in C/C++ despite the development and deployment of several defenses. Basic mitigations include Data Execution Prevention (DEP) [63] to stop code injection, stack canaries (GS) [22] to stop stack-based buffer overflows, and Address Space Layout Randomization (ASLR) [48] to probabilistically make code reuse attacks harder. These mitigations can be bypassed through, e.g., information leaks [28, 38, 42, 51] or code reuse attacks [13, 37, 56, 57, 66].

Advanced control-flow hijacking defenses such as Control-Flow Integrity (CFI) [11, 14, 41, 61] or shadow stacks/safe stacks [40] limit the set of allowed target addresses for indirect control-flow transfers. CFI mechanisms typically rely on static analysis to recover the Control-Flow Graph (CFG) of the application. These analyses over-approximate the allowed targets for each indirect dispatch location. At runtime, CFI checks determine if the observed target for each indirect dispatch location is within the allowed target set for that dispatch location as identified by the CFG analysis. Modern CFI mechanisms [41, 44, 45, 61] are deployed in, e.g., Google Chrome [60] or Microsoft Windows 10 and Edge [59].

However, CFI still allows the attacker control over the execution along two dimensions: first, the imprecision in the analysis enables the attacker to choose any of the targets in the set for each dispatch; second, data-only attacks allow an attacker to influence conditional branches arbitrarily. Existing attacks against CFI leverage manual analysis to construct exploits for specific applications along these two dimensions [16, 24, 29, 31, 53]. With CFI, exploits become highly program dependent as the set of gadgets is severely limited (due to the restrictions for indirect control-flow), exploits must therefore follow valid paths in the CFG. Finding a path along the CFG and satisfying its constraints is much more complex than simply finding the locations of gadgets. Finding attacks against advanced control-flow hijacking defenses is therefore predominantly a challenging manual process.

We present BOPC, an automatic framework to evaluate a program's remaining attack surface under strong control-flow hijacking mitigations. BOPC automates the task of finding an execution trace through a buggy program that executes arbitrary, attacker-specified behavior. BOPC compiles an "exploit" into a program trace, executing on top of the original program's Control-Flow Graph (CFG). To flexibly express exploit payloads, BOPC leverages a Turing-complete, high-level language: SPloit Language (SPL). To interact with the environment, SPL provides a rich API to call OS functions, direct access to memory, and an abstraction for hardware registers that allows a flexible mapping. BOPC takes as input an SPL payload and a starting point (e.g., found through fuzzing or manual analysis) and returns a trace through the program (encoded as a set of memory writes) that encodes the SPL payload.

The core component of BOPC is the mapping process through a novel code reuse technique we call *Block Oriented Programming*

Technical Report, West Lafayette, USA
2018. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

(BOP). First, BOPC translates the SPL payload into constraints for individual statements and, for each statement, searches for basic blocks in the target binary that satisfy these constraints (called *candidate blocks*). SPL keeps register assignments abstract from the underlying architecture. Second, BOPC infers a resource (register and state) mapping for each SPL statement, iterating through the set of candidate blocks and turning them into "*functional blocks*". Functional blocks can be used to execute a concrete instantiation of the given SPL statement. Third, BOPC constructs a trace that connects each functional block through *dispatcher blocks*. Since the mapping process is NP-hard, to find a solution in reasonable time BOPC first prunes the set of functional blocks per statement to constrain the search space and then uses a ranking based on the proximity of individual function blocks as a heuristic when searching for dispatcher gadgets.

We evaluate BOPC on 10 popular network daemons and setuid programs, demonstrating that BOPC can generate traces from a set of 13 test payloads. Our test payloads are both reasonable exploit payloads (e.g., calling `execve` with attacker-controlled parameters) as well as a demonstration of the computational capabilities of SPL (e.g., loops, or conditionals). Applications of BOPC go beyond an attack framework. We envision BOPC as a tool for defenders and software developers to highlight the *residual* attack surface of a program. For example, a developer can test whether a bug at particular statement enables a practical code reuse attack in their program to focus further bug detection. Overall, we present the following contributions:

- *Abstraction:* We introduce SPL, a C dialect that provides access to virtual registers and a rich API to call OS and other library functions, suitable for writing exploit payloads. SPL enables the necessary abstraction to scale to large applications.
- *Search:* Development of a *trace module* that allows execution of an arbitrary payload, written in SPL, using code of the target binary. The trace module considers strong defenses such as DEP, ASLR, shadow stack, and CFI alone or in combination. The trace module enables the discovery of viable mappings through a search process.
- *Evaluation:* Evaluation of our prototype demonstrates the generality of our mechanism and uncovers exploitable vulnerabilities where manual exploitation may have been infeasible. For 10 target programs, BOPC successfully generates exploit payloads and program traces to implement code reuse attacks for 13 SPL exploit payloads for 81% of the cases.

2 BACKGROUND AND RELATED WORK

Initially, exploits relied on simple code injection to execute arbitrary code. The deployment of Data Execution Prevention (DEP) [63] mitigated code injection and attacks moved to *reusing* existing code. The first code reuse technique, *return to libc* [26], simply reused existing libc functions. *Return Oriented Programming* (ROP) [56] extended code reuse to a Turing-complete technique. ROP locates small sequences of code which end with a return instruction, called "gadgets". Gadgets are connected by injecting the correct state, e.g., by preparing a set of invocation frames on the stack [56]. A number of code reuse variations followed [13, 19, 32], extending the approach from return instructions to arbitrary indirect control-flow transfers.

Several tools [30, 46, 52, 54] seek to automate ROP payload generation. However, the automation suffers from inherent limitations. Those tools fail to find gadgets in the target binary that do not follow the expected form "`inst1; inst2; ... retn;`". These tools search for a set of hard coded gadgets that form predetermined gadget chains. Instead of abstracting the required computation, the tools search for specific gadgets. If any gadget is not found or if a more complex gadget chain is needed, these tools degenerate to gadget dump tools, leaving the process of gadget chaining to the researcher who manually creates exploits from discovered gadgets.

The invention of code reuse attacks resulted in a plethora of new detection mechanisms based on execution anomalies and heuristics [20, 25, 35, 47, 50] such as frequency of return instructions. Such heuristics can often be bypassed [17].

While the aforementioned tools help to craft appropriate payloads, finding the vulnerability is an orthogonal process. Automatic Exploit Generation (AEG) [12] was the first attempt to automatically find vulnerabilities and generate exploits for them. AEG is limited in that it does not assume any defenses (such as the now basic DEP or ASLR mitigations). The generated exploits are therefore buffer overflows followed by static shellcode.

2.1 Control Flow Integrity

Control Flow Integrity [11, 14, 41, 61] (CFI) *prevents* control flow hijacking to arbitrary locations (and therefore code reuse attacks). CFI restricts the set of potential targets that are reachable from an indirect branch. While CFI does not stop the initial memory corruption, it validates the code pointer before it is used. CFI infers a CFG of the program to determine the allowed targets for each indirect control flow transfer. Before each indirect branch, the target address is checked to determine if it is a valid edge in the CFG, and if not an exception is thrown. This limits the freedom for the attacker, as she can only target a small set of targets instead of any executable byte in memory. For example, an attacker may overwrite a function pointer through a buffer overflow, but the function pointer is checked before it is used. Note that CFI targets the *forward edge*, i.e., virtual dispatch for C++ or indirect function calls for C. CFI is deployed in modern browsers (Google Chrome and Microsoft Edge) due to its success in preventing control-flow hijacking.

With CFI, code reuse attacks become harder, but not impossible [16, 29, 31, 53]. Depending on the application and strength of the CFI mechanism, CFI can be bypassed with Turing-complete payloads, which are often of high complexity to ensure compliance with the CFG. So far, these code-reuse attacks rely on manually constructed payloads.

Deployed CFI implementations [41, 44, 45, 49, 61] use a static over-approximation of the CFG – based on method prototypes and class hierarchy. PittyPat [27] and PathArmor [64] introduce path sensitivity that evaluates partial execution paths. Newton [65] introduced a framework that reasons about the strength of defenses, including CFI. Newton exposes indirect pointers (along with their

allowed target set) that are reachable (i.e., controllable by an adversary) through given entry points. While Newton displays all usable "gadgets", it cannot stitch them together and effectively is a CFI-aware ROP gadget search tool that helps an analyst to manually construct an attack.

2.2 Shadow Stacks

While CFI protects *forward* edges in the CFG (i.e., function pointers or virtual dispatch), a shadow stack orthogonally protects *backward* edges (i.e., return addresses) [23]. Shadow stacks keep a protected copy (called *shadow*) of all return addresses on a separate, protected stack. Function calls store the return address both on the regular stack and on the shadow stack. When returning from a function, the mitigation checks for equivalence and reports an error if the two return addresses do not match. The shadow stack itself is assumed to be at a protected memory location to keep the adversary from tampering with it. Shadow stacks enforce stack integrity and protect from any control-flow hijack attack against the backward edge.

2.3 Data-only Attacks

While CFI mitigates most code reuse attacks, CFI cannot stop data-only attacks. Manipulating a program's *data* can be enough for a successful exploitation. Data-only attacks target the program's data rather than the execution flow. E.g., having full control over the arguments to `execve()` suffices for arbitrary command execution. Also, data in a program may be sensitive: consider overwriting the `uid` or a variable like `is_admin`. Data-only attacks were generalized and defined formally as *Data Oriented Programming* (DOP) [34]. Existing DOP attacks rely on an analyst to identify sensitive variables for manual construction.

Similarly to CFI, it is possible to build the *Data Flow Graph* of the program and apply *Data Flow Integrity* (DFI) [18] to it. However, to the best of our knowledge, there are no practical DFI-based defenses due to prohibitively high overhead of data-flow tracking.

In comparison to existing data-only attacks, BOPC automatically generates payloads based on a high-level programming language. The payloads follow the valid CFG of the program but not its Data Flow Graph.

3 ASSUMPTIONS AND THREAT MODEL

Our threat model consists of a binary with a known memory corruption vulnerability that is protected with the state-of-the-art control-flow hijack mitigations, such as Control Flow Integrity (CFI) along with a Shadow Stack. Furthermore, the binary is also hardened with Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR) and Stack Canaries (GS).

We assume that the target binary has an arbitrary memory write vulnerability. That is, the attacker can write *any* value to *any* (writable) address. We call this an *Arbitrary memory Write Primitive* (AWP). To bypass probabilistic defenses such as ASLR, we assume that the attacker has access to an information leak, i.e., a vulnerability that allows her to read *any* value from *any* memory address. We call this an *Arbitrary memory Read Primitive* (ARP).

We also assume that there exists an entry point, i.e., a location that the program reaches naturally and occurs after all AWPs and ARPs have been completed. This can be an attacker-controlled code pointer where the control flow is hijacked. Determining an entry point is considered to be part of the vulnerability discovery process. Thus, finding this entry point is orthogonal to our work.

Note that these assumptions are in line with the threat model of control-flow hijack mitigations that aim to prevent attackers from exploiting arbitrary read and write capabilities. These assumptions are also practical. Orthogonal bug finding tools such as fuzzing often discover arbitrary memory accesses that can be abstracted to the required arbitrary read and writes with an entry point right after the AWP. Furthermore, these assumptions map to real bugs. Web servers, such as nginx, spawn threads to handle requests and a bug in the request handler can be used to read or write an arbitrary memory address. Due to the request-based nature, the adversary can repeat this process multiple times. After the completion of the state injection, the program follows an alternate and disjoint path to trigger the injected payload.

These assumptions enable BOPC to inject the payload into the program, modifying its execution state and starting the payload execution from the given entry point. BOPC assumes that the AWP and ARP may be triggered multiple times to modify the execution state of the target binary. After the state modification completes, the SPL payload executes without further changes in execution state. This separates SPL execution into two phases: state modification and execution. The AWP/ARP allow state modification, BOPC infers the required state change to execute the SPL payload.

4 DESIGN

Figure 1 shows how BOPC automates the analyst tasks necessary to leverage AWPs and/or ARPs to produce a useful exploit in the presence of strong defenses, including CFI. First, BOPC provides an exploit programming language, called *SPloit Language* (SPL), that enables analysts to define exploits independent of the target program or underlying architecture. Second, to automate how analysts find gadgets that implement SPL statements that comply with CFI, BOPC finds basic blocks from the target program that implement individual SPL statements, called *functional blocks*. Third, to enable analysts to chain basic blocks together in a manner that complies with CFI and shadow stacks, BOPC searches the target program for sequences of basic blocks that connect pairs of neighboring functional blocks, which we call *dispatcher blocks*. Fourth, BOPC simulates the BOP chain to produce a payload that implements that SPL payload from a chosen AWP.

The BOPC design builds on two key ideas: Block Oriented Programming and Block Constraint Summaries. First, defenses such as CFI, impose stringent restrictions on transitions between gadgets, so we no longer have the flexibility of setting the instruction pointer

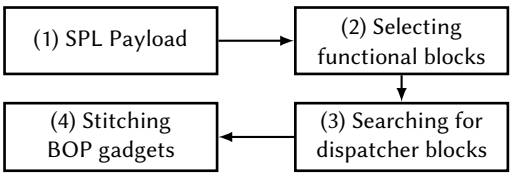

Figure 1: Overview of BOPC's design.

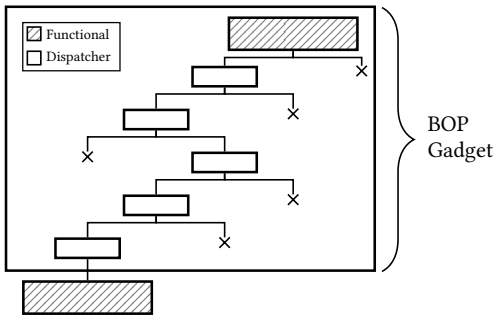

Figure 2: BOP gadget structure. The *functional* part consists of a single basic block that executes an SPL statement. Two functional blocks are chained together through a series of *dispatcher* blocks, without clobbering the execution of the previous functional blocks.

to arbitrary values. Instead, BOPC implements *Block Oriented Programming* (BOP), which constructs exploit programs called *BOP chains* from basic block sequences in the valid CFG of a target program. Note that our CFG encodes both forward edges (protected by CFI) and backward edges (protected through a shadow stack). For BOP, gadgets are no longer arbitrary sequences of instructions ending in an indirect control-flow transfer, but chains of entire basic blocks (sequences of instructions that end with a direct *or* indirect control-flow transfer), as shown in Figure 2. A BOP chain consists of a sequence of *BOP gadgets* where each BOP gadget is: one *functional block* that implements a statement in an SPL payload and zero or more *dispatcher blocks* that connect the functional block to the next BOP gadget in a manner that complies with the CFG.

Second, BOPC abstracts each basic block from individual instructions operating on specific registers into *Block Constraint Summaries*, enabling blocks to be employed in a variety of different ways. That is, a single block may perform multiple functional and/or dispatching operations by utilizing different sets of registers for different operations, As an example, a basic block that modifies register `rdx` unintentionally, is clobbering if `rdx` is part of the register mapping, or a dispatcher block if it is not. In addition, BOPC leverages abstract block constraint summaries to apply blocks in multiple contexts. At each stage in the development of a BOP chain, the blocks that may be employed next in the CFG as dispatcher blocks to connect two functional blocks depend on the block summary constraints for each block. There are two cases: either the candidate dispatcher block's summary constraints indicate that it will modify the register state set by the functional blocks, called the *SPL state*, or it will not, enabling the computation to proceed without disturbing the effects of the functional blocks. A block that modifies a current SPL state is said to be a *clobbering block* for that state. Block summary constraints enable identification of clobbering blocks at each point in the search.

An important distinction between BOP and conventional ROP (and variants) is that the problem of computing BOP chains is NP-hard, as proven in Appendix B. Conventional ROP assumes that indirect control-flows may target any executable byte (or a subset thereof) in memory while BOP must follow a legal path through the CFG for any chain of blocks, which motivates the need for tooling support.

4.1 Expressing Payloads

To start a search for exploits, analysts must identify what constitutes a useful exploit. In the past, analysts likely have had a small number of exploit types in mind to guide manual search from gadgets, but the specific nature of the exploit depends on the low-level details of the target program and processor architecture. Thus, writing exploits has little benefit since they will differ depending on the gadgets available in the target program. Previous automated approaches for exploit generation were designed with a specific type of exploit in mind, so they built the exploit specifications into their tools procedurally.

When searching for exploits against strong defenses automatically, such ad hoc approaches will not suffice. Knowledge of the gadgets necessary to perform an exploit cannot be built into the exploit generation program because the way each exploit will be implemented by blocks and the way that such blocks may be chained together varies from target program to target program. In addition, we want to enable analysts to generate exploit payloads for target programs built for different processor architectures without them having to be an expert in that processor architecture.

To address this problem, BOPC provides a programming language, called *SPloit Language* (SPL) that allows analysts to express exploit payloads in a compact high-level language that is independent of target programs or processor architectures. SPL is a dialect of C. Table 1 shows some sample payloads. Overall, SPL has the following features:

- It is Turing-complete;
- It is architecture independent;
- It is close to a well known, high level language.

Compared to existing exploit development tools [30, 46, 52, 54], the architecture independence of SPL has important advantages. First, the same payload can be executed under different ISAs or operating systems. Second, SPL uses a set of *virtual registers*, accessed through reserved volatile variables. Virtual registers increase flexibility, which in turn increases the chances of finding a solution. That is, when payload uses a virtual register, any general purpose register (16 for x86-64) may be used.

To interact with the environment, SPL defines a concise API to access OS functionality. Finally, SPL supports conditional and unconditional jumps to enable control-flow transfers to arbitrary locations. This feature makes SPL a Turing-complete language, proven in Appendix C. The complete language specifications are shown in Appendix A in Extended Backus–Naur form (EBNF).

The environment for SPL differs from that of conventional languages. Instead of running code directly on a CPU, our compiler encodes the payload as a mapping of instructions to functional blocks. That is, the underlying runtime environment is the target binary and its program state, where payloads are executed as side effects of the underlying binary.

Simple loop

```
void payload() {
  __r0 = 0;

LOOP:
  __r0 += 1;
  if (__r0 != 128)
    goto LOOP;

  returnto 0x446730;
}
```

Spawn a shell

```
void payload() {
  string prog = "/bin/sh\0";
  int64 *argv = {&prog, 0x0};

  __r0 = &prog;
  __r1 = &argv;
  __r2 = 0;

  execve(__r0, __r1, __r2);
}
```

Table 1: Examples of SPL payloads.

4.2 Selecting functional blocks

To generate a BOP chain for an SPL payload, BOPC must find a sequence of blocks that implement each statement in the SPL payload, which we call *functional blocks*. The process of building BOP chains starts by identifying functional blocks per SPL statement.

Conceptually, BOPC must compare each block to each SPL statement to determine if the block can implement the statement. However, blocks are in terms of machine code and SPL statements are high-level program statements. To provide flexibility for matching blocks to SPL statements, BOPC computes *block constraint summaries*, which define the possible impacts that the block would have on SPL state. Block constraint summaries provide flexibility in matching blocks to SPL statements because there are multiple possible mappings of SPL statements and their virtual registers to the block and its constraints on registers and state.

The constraint summaries of each basic block is obtained by *isolating* and *symbolically* executing it. The effect of symbolically executing a basic block creates a set of constraints, mapping input to the resultant output. Such constraints may refer to registers, memory locations, and external operations (e.g., library calls).

To find a match between a block and an SPL statement the block must perform all the operations required for that SPL statement. More specifically, the constraints of the basic block must be a superset of the operations required to implement the SPL statement.

4.3 Finding BOP gadgets

BOPC computes a *set* of all *potential* functional blocks for each SPL statement or halts if any statement has no blocks. To stitch functional blocks, BOPC must select one functional block and a sequence of dispatcher blocks that reach the next functional block in the payload. The combination of a functional block and its dispatcher blocks is called a *BOP gadget*, as shown in Figure 2. To build a BOP chain, BOPC must select *exactly* one functional block from each set and find the appropriate dispatcher blocks to connect all functional blocks together.

However, dispatcher paths between two functional blocks may not exist. Either because there is no legal path in the CFG between them, or the execution flow cannot naturally reach the next block due to un-satisfiable runtime constraints. This constraint imposes limits on functional block selection, as the existence of a dispatcher path depends on the *previous* BOP gadgets.

BOP gadgets are *volatile*: gadget feasibility changes based on the execution state (i.e., context) of the target binary. This concept is illustrated in Figure 3. The problem of selecting a suitable sequence of functional blocks, such that a dispatcher path exists between

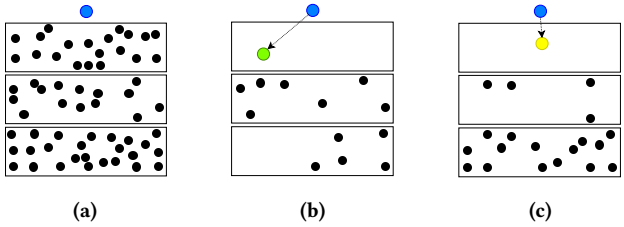

Figure 3: Visualisation of BOP gadget volatility, rectangles: SPL statements, dots: functional blocks (a). Connecting any two statements through dispatcher blocks constrains remaining gadgets (b), (c).

every possible control flow transfer in the SPL payload, is NP-hard (as we prove in Appendix B), and we are not aware of an approximative algorithm.

As the problem is not solvable in polynomial time, we propose several heuristics and optimizations to find solutions in reasonable amounts of time. BOPC leverages basic block *proximity* as a metric to "rank" dispatcher paths and organizes this information into a special data structure, called a *delta graph* that provides an efficient way to probe potential sequences of functional blocks.

4.4 Searching for dispatcher blocks

While each functional block executes a statement, BOPC must chain multiple functional blocks together to execute the SPL payload. Functional blocks are connected through zero to L blocks that may not clobber the SPL state computed thusfar. Finding such non-clobbering blocks that transfer control from one functional statement to another is challenging as each additional block increases the constrains and path dependencies. We propose a graph data structure, the *delta graph* to represent the state of the search for dispatcher blocks. The delta graph stores, for each functional block for each SPL, statement the shortest path to the next candidate block. Stitching arbitrary sequences of statements is NP-hard as each selected path between two functional statements influences the availability of further candidate blocks or paths, we therefore leverage the delta graph to try *likely* candidates first.

The intuition behind the *proximity* of functional blocks is that shorter paths result in simpler and satisfiable constraints. Although this metric is a *heuristic* (see Table 2 for a counter example), our evaluation (Section 6) shows that it works well in practice.

The delta graph enables quick elimination of sets of functional blocks that are highly unlikely to have dispatcher blocks and thus constitute a BOP gadget. For instance, if there is no valid path in the CFG between two functional blocks (e.g., if execution has to traverse the CFG "backwards"), no dispatcher will exist and therefore, these two functional blocks cannot be part of the solution.

The delta graph, is a multi-partite, directed graph which has a set of functional block nodes for *every* payload statement. An edge between two functional blocks represents the *minimum* number of executed basic blocks to move from one functional block to the other, while *avoiding* clobbering blocks. See Figure 7 for an example.

Indirect control-flow transfers pose an interesting challenge when calculating the shortest path between two basic blocks in

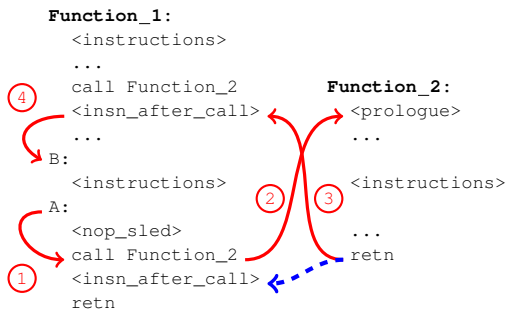

Figure 4: Existing shortest path algorithms are unfit to measure proximity in the CFG. Consider the shortest path from `A` to `B`. A context-unaware shortest path algorithm will mark the red path as solution, instead of following the blue arrow upon return from `Function_2`, it follows the red arrow (3).

Long path with simple constraints	*Short path with complex constraints*

`a, b, c, d, e = input();`
`// point A`
`if (a == 1) {`
`  if (b == 2) {`
`  if (c == 3) {`
`  if (d == 4) {`
`  if (e == 5) {`
`  // point B`
`  ...`
	`a = input();`
	
	`X = sqrt(a);`
	`Y = log(a*a*a - a)`
	
	`// point A`
	`if (X == Y) {`
	`  // point B`
	`  ...`

Table 2: A counterexample that demonstrates why proximity between two functional blocks can be inaccurate. Left, we can move from point A to point B even if they are 5 blocks apart from each other. Right, it is much harder to satisfy the constrains and to move from A to B, despite the fact that A and B are only 1 block apart.

a CFG: while they statically allow multiple targets, at runtime they are context sensitive and only have one concrete target. Our context sensitive shortest path algorithm is a *recursive* version of Dijkstra's [21] shortest path algorithm. We start with regular shortest path, assuming that every edge on the CFG has cost 1. Each time we encounter a basic block that ends with a call instruction, we recursively run a new shortest path algorithm, starting from the calling function. If the destination basic block is inside the caller function, then the shortest path is the addition of the two individual shortest paths (from the beginning to the function's entry point and from there to the target block). Otherwise, we calculate the *shortest path* from the function's entry point to the closest return and use this value as an edge weight to the callee. Our algorithm uses a *call stack* to keep track of all visited functions to avoid infinite loops in case of recursive functions. Finally, our algorithm avoids *any* basic block that are marked as clobbering.

After creation of the delta graph, our algorithm selects *exactly* one node (i.e., functional block) from each set (i.e., payload statement), to *minimize* the total weight of the resulting *induced subgraph* [1]. This selection of functional blocks is considered to be the most likely to give a solution, so the next step is to find the exact dispatcher blocks and create the BOP gadgets for the SPL payload.

[1]The *induced subgraph* of the delta graph is a subgraph of the delta graph with one node (functional block) for each SPL statement and with edges that represent their shortest available dispatcher block chain.

4.5 Stitching BOP gadgets

The minimum induced subgraph from the previous step determines a set of functional blocks that may be stitched together into an SPL payload. This set of functional blocks is close to each other, making satisfiable dispatcher paths more likely.

To find a dispatcher path between two functional blocks, BOPC leverages *concolic execution* [55] (symbolic execution along a given path). Along the way, it collects the required constraints that are needed to lead the execution to the next functional block. Symbolic execution engines [15, 58] translate basic blocks into sets of constraints and use Satisfiability Modulo Theories (SMT) to find satisfying assignments for these constraints; symbolic execution is therefore NP-complete. Starting from the (context sensitive) shortest path between the functional blocks, BOPC *guides* the symbolic execution engine, collecting the corresponding constraints.

To construct an SPL payload from a BOP chain, BOPC launches concolic execution from the first functional block in the BOP chain, starting with an empty state. At each step BOPC tries the first K shortest dispatcher paths until it finds one that reaches the next functional block (the edges in the minimum induced subgraph indicate which is the "next" functional block). The corresponding constraints are added to the current state. The search therefore *incrementally* adds BOP gadgets to the execution state. When a functional block represents a conditional SPL statement, its node in the induced subgraph contains two outgoing edges (i.e., the execution can transfer control to two different statements). However during the concolic execution, the algorithm does not know which one will be followed, it *clones* the current state and independently follows both branches, exactly like symbolic execution [15].

Reaching the last functional block, BOPC checks whether the constraints have a satisfying assignment and forms an exploit payload. Otherwise, it falls back and tries the next possible set of functional blocks. To repeat that execution on top of the target binary, these constraints are concretized and translated into a memory layout that will be initialized through AWP in the target binary.

5 IMPLEMENTATION

Our open source prototype, BOPC, is implemented in Python and consists of approximately 14,000 lines of code. BOPC requires three distinct inputs:

- The exploit payload expressed in SPL,
- The vulnerable application on top of which the SPL payload runs,
- The entry point in the vulnerable application, which is a location that program reaches naturally and occurs after all memory writes have been completed.

The output of BOPC is a sequence of $(address, value, size)$ tuples that describe how the memory should be modified during the state modification phase (Section 3) to execute the payload. Optionally, it may also require some additional $(stream, value, size)$ tuples that describe what input should be given on any potentially open "streams" (file descriptors, sockets, stdin) that the attacker controls during the execution of the payload.

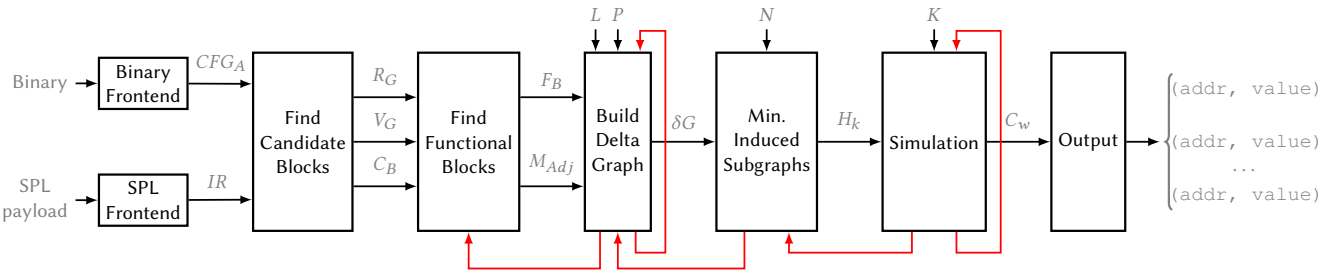

Figure 5: High level overview of the BOPC implementation. The red arrows indicate the iterative process upon failure. CFG_A: CFG with basic block abstractions added, IR: Compiled SPL payload R_G: Register mapping graph, V_G: All variable mapping graphs, C_B: Set of candidate blocks, F_B: Set of functional blocks, M_{Adj}: Adjacency matrix of SPL payload, δG: Delta graph, H_k: Induced subgraph, C_w: Constraint set. L: Maximum length of continuous dispatcher blocks. P: Upper bound on payload "shuffles", N: Upper bound on minimum induced subgraphs, K: Upper bound on shortest paths for dispathers,

A high level overview of BOPC is shown in Figure 5. Our algorithm is *iterative*; that is, in case of a failure, the red arrows, indicate which module is executed next.

5.1 Binary Frontend

The Binary Frontend, lifts the target binary into an intermediate representation that exposes the application's CFG. Operating directly on basic blocks is cumbersome and heavily dependent on the Application Binary Interface (ABI). Instead, we translate each basic block into a *block constraint summary*. Abstraction leverages symbolic execution [39] to "summarize" the basic block into a set of constraints encoding changes in registers and memory, and any potential system, library call, or conditional jump at the end of the block – generally any *effect* that this block has on the program's state. BOPC executes each basic block in an isolated environment, where every action (such as accesses to registers or memory) is monitored. Therefore, instead of working with the instructions of each basic block, BOPC utilizes its abstraction for all operations. The abstraction information for every basic block is added to the CFG, resulting in CFG_A.

5.2 SPL Frontend

The SPL Frontend translates the exploit payload into a graph-based Intermediate Representation (IR) for further processing. To increase the flexibility of the mapping process, statements in a sequence may be executed out-of-order. For each statement sequence we build a *dependence graph* based on a customized version of Kahn's [36] topological sorting algorithm, to infer all groups of independent statements. Independent statements in a subsequence are then turned into a set of statements which can be executed out-of-order. This results in a set of equivalent payloads that are essentially permutations of the original. Our goal is to find a solution for *any* of them.

5.3 Locating candidate block sets

SPL is an high level language that hides the underlying ABI. Therefore, BOPC looks for potential ways to "map" the SPL environment to the underlying ABI. The key insight in this step, is to find all possible ways to map the individual elements from the SPL environment to the ABI (though *candidate blocks*) and then iteratively selecting valid subsets from the ABI to "simulate" the environment of the SPL payload.

Once the CFG_A and the IR are generated, BOPC searches for and marks *candidate* basic blocks, as described in Section 4.2. For a block to be candidate, it must "semantically match" with one (or more) payload statements. Table 3 shows the matching rules. Note that variable assignments, unconditional jumps, and returns do not require a basic block and therefore are excluded from the search.

All statements that assign or modify registers require the basic block to apply the same operation on undetermined hardware registers. For function calls, the requirement for the basic block is to invoke the same call, either as a system call or as a library call. Note that the calling convention exposes the register mapping.

Upon a successful matching, BOPC builds the following data structures:

- R_G, the *Register Mapping Graph* which is a bipartite undirected graph. The nodes in the two sets represent the virtual and hardware registers respectively. The edges represent potential associations between virtual and hardware registers.
- V_G, the *Variable Mapping Graph*, which is very similar to R_G, but instead associates payload variables to underlying memory addresses. V_G is unique for every edge in R_G i.e.:

$$\forall(\_\_r_\alpha, reg_\gamma) \in R_G \; \exists! \; V_G^{\alpha\gamma} \tag{1}$$

- D_M, the *Memory Dereference Set*, which has all memory addresses, that are dereferenced and their values are loaded into registers. Those addresses can be symbolic expressions (e.g., `[rbx + rdx*8]`), and therefore we do not know the concrete address they point to, until execution reaches them.

After this step, each SPL statement has a *list* of candidate blocks. Note that a basic block can be candidate for multiple statements. If for some statement there are no candidate blocks, the algorithm halts and reports that the program cannot be synthesized.

Statement	Form	Abstraction	Actions			Example
Register Assignment	$\_\_r_\alpha = C$	$reg_\gamma \leftarrow C$	$R_G \cup \{(\_\_r_\alpha, reg_\gamma)\}$	–		`movzx rax, 7h`
		$reg_\gamma \leftarrow *A$		$D_M \cup \{A\}$		`mov rax, ds:fd`
	$\_\_r_\alpha = \&V$	$reg_\gamma \leftarrow C,\ C \in R \wedge W$		$V_G^{\alpha\gamma} \cup \{(V, A)\}$	–	`lea rcx, [rsp+20h]`
		$reg_\gamma \leftarrow *A$			$D_M \cup \{A\}$	`mov rdx, [rsi+18h]`
Register Modification	$\_\_r_\alpha \odot = C$	$reg_\gamma \leftarrow reg_\gamma \odot C$	$R_G \cup \{(\_\_r_\alpha, reg_\gamma)\}$			`dec rsi`
Memory Read	$\_\_r_\alpha = *\_\_r_\beta$	$reg_\gamma \leftarrow *reg_\delta$	$R_G \cup \{(\_\_r_\alpha, reg_\gamma),\ (\_\_r_\beta, reg_\delta)\}$			`mov rax, [rbx]`
Memory Write	$*\_\_r_\alpha = \_\_r_\beta$	$*reg_\gamma \leftarrow reg_\delta$				`mov [rax], [rbx]`
Call	$call(\_\_r_\alpha,\ \_\_r_\beta,\ ...)$	`Ijk_Call` to *call*	$R_G \cap \{(\_\_r_\alpha, \%rdi),\ (\_\_r_\beta, \%rsi),\ ...\}$			`call execve`
Conditional Jump	*if* $(\_\_r_\alpha \odot = C)$ *goto LOC*	`Ijk_Boring` $\wedge$ *condition* $= reg_\gamma \odot C$	$R_G \cup \{(\_\_r_\alpha, reg_\gamma)\}$			`test rax, rax` `jnz LOOP`

Table 3: Semantic matching of SPL statements to basic blocks. Abstraction indicates the requirements that the basic block abstraction needs to have to match the SPL statement in the Form. Upon a match, the appropriate Actions are taken. $\_\_r_\alpha$, $\_\_r_\beta$: Virtual registers, reg_γ, reg_δ: Hardware registers, C: Constant value, V: SPL variable, A: Memory address, R_G: Register mapping graph, V_G: Variable mapping graph, D_M: Dereferenced Addresses Set, `Ijk_Call`: A call to an address, `Ijk_Boring`: A normal jump to an address.

5.4 Identifying functional block sets

After determining the set of candidate blocks, C_B, BOPC *iteratively* identifies, for each SPL statement, which candidate blocks can serve as functional blocks, i.e., the blocks that perform the operations. This step determines for each candidate block if there is a resource mapping that satisfies the block's constraints.

BOPC identifies the *concrete* set of hardware registers and memory addresses that execute the desired statement. A successful mapping identifies candidate blocks that serve as functional blocks.

To find the hardware-to-virtual register association, BOPC searches for a *maximum bipartite matching* [21] in R_G. If such a mapping does not exists, the algorithm halts. The selected edges indicate the set of V_G graphs that are used to find the variable-to-address association (see Section 5.3, there can be a V_G for every edge in R_G). Then for every V_G the algorithm repeats the same process to find another maximum bipartite matching.

This step determines, for each statement, which concrete registers and memory addresses are reserved. Merging this information with the set of candidate blocks removes clobbering blocks, i.e., any candidate blocks that are unsatisfiable.

However, the previous mapping may not be unique (there may be other sets of functional blocks). If the current mapping does not lead to a solution, the algorithm revisits an *alternate* mapping iteratively. The algorithm enumerates *all* maximum bipartite matchings [62], trying them one by one. If no matching leads to a solution, the algorithm halts.

5.5 Selecting functional blocks

Given the functional block set F_B, this step searches for a set that executes all payload statements. The goal is to select *exactly* one functional block for every IR statement and find dispatcher blocks to chain them together. BOPC builds the *delta graph* δG, described in Section 4.4.

Once the delta graph is generated, this step locates the *minimum induced subgraph*, which is the exact set of functional blocks that execute the payload. If the minimum induced subgraph does not result in a solution, the algorithm tries the second shortest subgraph, and so on. As an exponential number of subgraphs may exist, this step limits the search to the N minimum.

If the resulting delta graph does not lead to a solution, this step "shuffles" out-of-order payload statements, see Section 5.2, and builds a new delta graph. Note that the number of different permutations may be exponential. Therefore, our algorithm sets an *upper bound* P on the number of tried permutations.

Each permutation results in a different yet semantically equivalent SPL payload, so the CFG of the payload (i.e., the *Adjacency Matrix*, M_{Adj} needs to be recalculated.

5.6 Discovering dispatcher blocks

The simulation phase takes the individual functional blocks (contained in the minimum induced subgraph H_k) and tries to find the appropriate dispatcher blocks, to compose the BOP gadgets. It returns a set of memory assignments for the corresponding dispatcher blocks, or an error indicating un-satisfiable constraints for the dispatchers.

BOPC is called to find a dispatcher path for *every* edge in the minimum induced subgraph. That is, we need to simulate every control flow transfer in the adjacency matrix, M_{Adj} of the SPL payload. However, dispatchers are *built* on the execution state, so BOP gadgets must be stitched with the respect to the execution flow which starts from the entry point.

Finding dispatcher blocks relies on concolic execution. Our algorithm utilizes functional block proximity as a metric for dispatcher path quality. However, it cannot predict which constraints will take exponential time to solve (in practice we set a timeout). Therefore concolic execution selects the K (context sensitive) shortest paths as dispatcher blocks and tries them (starting from the shortest) until one leads to a set of satisfiable constraints. It turns that this metric works very well in practice even for small values of K (e.g., 8).

Thus, we find the shortest path between two functional blocks (i.e., the BOP dispatcher), and we instruct the symbolic execution engine to follow it. Alternatively, the simulation tries the second shortest one and so on. This is similar to the *k-shortest path* [67] algorithm used for the delta graph.

When simulation starts it also initializes any SPL variables at the locations that are reserved during the variable mapping (Section 5.4). These addresses are marked as *immutable*, so any unintended modification raises an exception which stops this iteration.

In Table 3, we introduce the set of *Dereferenced Addresses*, D_M, which is the set of memory addresses whose contents are loaded into registers. Simulation cannot obtain the exact location of a symbolic address (e.g., `[rax + 4]`) until the block is executed and the register has a concrete value. Before simulation reaches a functional block, it concretizes any symbolic addresses from D_M and initializes the memory cell accordingly. If that memory cell has already been set, any initialization *prior* to the entry point cannot persist. That is, BOPC cannot leverage an AWP to initialize this memory cell and the iteration fails. If a memory cell has been used in the constraints, its concretization can make constraints unsatisfiable and the iteration may fail.

Simulation traverses the minimum induced subgraph, and *incrementally* extends the execution state from one BOP gadget to the next. Encountering a conditional statement (i.e., a functional block has two outgoing edges), BOPC *clones* the current state and continues building the trace for both paths independently, in the same way that a symbolic execution engine handles conditional statements. When a path reaches a functional block that was already visited, it gracefully terminates. At the end, we collect all those states and check whether the constraints of these paths are satisfiable or not. If so, we have a solution.

5.7 Synthesizing exploit from execution trace

If the simulation module returns a solution, the final step is to encode the execution trace as a set of memory writes in the target binary. The constraint set C_w collected during simulation reveal a memory layout that leads to an execution flow across functional block according to the minimum induced subgraph.

Concretizing the constraints for all participating conditional variables at the *end* of the simulation can result in incorrect solutions, as shown in the following example:

```
a = input();
if (a > 10 && a < 20) {
    a = 0;
    /* target block */
}
```

The symbolic execution engine concretizes the symbolic variable assigned to `a`, upon assignment. When execution reaches "target block", `a` is 0, which is contradicts the precondition to reach the target block. Hence, BOPC needs to resolve the constraints *on the fly*, rather than at the end of the simulation.

Therefore, this step is done alongside the simulation, carefully monitoring all variables and concretizing them at the right moment. More specifically, memory locations that are accessed for first time, are assigned a symbolic variable. Whenever a memory write occurs, a check ensures that the initial symbolic variable persists in the new symbolic expression. If not, BOPC concretizes it, adding the concretized value to the set of memory writes.

There are also some symbolic variables that do *not* participate in the constraints, but are used as pointers. These variables are concretized to point to a writable location to avoid segmentation faults outside of the simulation environment.

Finally, it is possible for registers or external symbolic variables (e.g., data from stdin, sockets or file descriptors) to be part of the constraints. BOPC executes a similar translation for the registers and any external input, as these are inputs to the program that are usually also controlled by the attacker.

6 EVALUATION

To evaluate BOPC, we leverage a set of 10 applications with known memory corruption CVEs, listed in Table 4. These CVEs correspond to arbitrary memory writes [16, 33, 34], fulfilling our AWP primitive requirement. Table 4 contains the total number of all functional blocks for each application. Although there are many functional blocks, the difficulty of finding stitchable dispatcher blocks makes a significant fraction of them unusable.

Basic block abstraction is a time consuming process – especially for applications with large Control-Flow Graphs – whose results do not change across iterations. As a performance optimization, BOPC caches the resulting abstractions of the Binary Frontend (Figure 5) to a file and loads them for each search, thus avoiding the startup overhead listed in Table 4.

To demonstrate the effectiveness of our algorithm, we chose a set of 13 representative SPL payloads, shown in Table 5. Our goal is to "map and run" each of these payloads on top each of the vulnerable applications. Table 6 shows the results of running each payload on. BOPC successfully finds a mapping of memory writes to encode an SPL payload as a set of side effects executed on top of the applications for 105 out of 130 cases, approximately 81%. In each case, the memory writes are sufficient to reconstruct the payload execution by strictly following the CFG without violating a strict CFI policy or stack integrity.

Table 6 shows that applications with large CFGs result in higher success rates, as they encapsulate a "richer" set of BOP gadgets. Achieving truly infinite loops is hard in practice, as most of the loops in our experiments involve some loop counter that is modified in each iteration. This iterator serves as an index to dereference an array. By falsifying the exit condition through modifying loop variables (i.e., the loop infinite), the program eventually terminates with a segmentation fault, as it tries to access memory outside of the current segment. Therefore, even though the loop would run forever, an external factor (segmentation fault) causes it to stop. BOPC aims to address this issue, by simulating the same loop multiple times. However, finding a truly infinite loop, requires BOPC to simulate it an infinite number of times, which is infeasible. For some cases, we managed to verify that the accessed memory inside the loop is bounded and therefore the solution truly is an infinite loop. Otherwise the loop is *arbitrarily bounded* with the upper bound set by an external factor.

For some payloads, BOPC was unable to find an exploit trace. This is is either due to imprecision of our algorithm, or because no solution exists for the written SPL payload. We can alleviate the first failure by increasing the upper bounds and the timeouts in our configuration. Doing so, makes BOPC search more exhaustively at the cost of search time.

The non-existence of a solution is an interesting problem, as it exposes the *limitations* of the vulnerable application. This type of failure is due to the "structure" of the application's CFG, which

Vulnerable Application			CFG		Time	Total number of functional blocks						
Program	**Vulnerability**	**Prim.**	**Nodes**	**Edges**	**(m:s)**	**RegSet**	**RegMod**	**MemRd**	**MemWr**	**Call**	**Cond**	**Total**
ProFTPd	CVE-2006-5815 [5]	AW	27,087	49,862	10:08	40,143	387	1,592	199	77	3,029	45,427
nginx	CVE-2013-2028 [9]	AW	24,169	44,645	12:36	31,497	1,168	1,522	279	35	3375	37,876
sudo	CVE-2012-0809 [8]	FMS	3,399	6,267	01:14	5,162	26	157	18	45	307	5715
orzhttpd	BugtraqID 41956 [7]	FMS	1,354	2,163	00:27	2,317	9	39	8	11	89	2473
wuftdp	CVE-2000-0573 [1]	FMS	8,899	17,092	03:22	14,101	62	274	11	94	921	15,463
nullhttpd	CVE-2002-1496 [3]	AW	1,488	2,701	00:27	2,327	77	54	7	19	125	2,609
opensshd	CVE-2001-0144 [2]	AW	6,688	12,487	01:53	8,800	98	214	19	63	558	9,752
wireshark	CVE-2014-2299 [10]	AW	74,186	162,111	29:41	12,4053	639	1,736	193	100	4555	131276
apache	CVE-2006-3747 [4]	AW	18,790	34,205	10:22	33,615	212	490	66	127	1,768	36,278
smbclient	CVE-2009-1886 [6]	FMS	166,081	351,309	82:25	265,980	1,481	6,791	951	119	28,705	304,027

Table 4: Vulnerable applications. The *Prim.* column indicates the primitive type (`AW` = Arbitrary Write, `FMS` = ForMat String). *Time* is the amount of time needed to generate the abstractions for every basic block. *Functional blocks* show the total number for each of the statements (*RegSet* = Register Assignments, *RegMod* = Register Modifications, *MemRd* = Memory Load, *MemWr* = Memory Store, *Call* = system/library calls, *Cond* = Conditional Jumps). Note that the number of call statements is small because we are targeting a predefined set of calls. Also note that MemRd statements are a subset of RegSet statements.

Payload	Description	$\|S\|$	flat?
regset4	Initialize 4 registers with arbitrary values	4	✓
regref4	Initialize 4 registers with pointers to arbitrary memory	8	✓
regset5	Initialize 5 registers with arbitrary values	5	✓
regref5	Initialize 5 registers with pointers to arbitrary memory	10	✓
regmod	Initialize a register with an arbitrary value and modify it	3	✓
memrd	Read from arbitrary memory	4	✓
memwr	Write to arbitrary memory	5	✓
print	Display a message to stdout using `write`	6	✓
execve	Spawn a shell through `execve`	6	✓
abloop	Perform an arbitrarily long bounded loop utilizing regmod	2	✗
infloop	Perform an infinite loop that sets a register in its body	2	✗
ifelse	An if-else condition based on a register comparison	7	✗
loop	Conditional loop with register modification	4	✗

Table 5: SPL payloads. Each payload consists of $|S|$ statements. Payloads with *flat* delta graphs (i.e., have no jump statements), are marked with ✓.

prevents BOPC from finding a trace for an SPL payload. A solution may not exist due to one the following:

(1) There are not enough candidate blocks.
(2) There are no valid register / variable mappings.
(3) There are not enough functional blocks or no valid paths between functional blocks.
(4) The constraints between blocks are un-satisfiable or symbolic execution raised aa timeout.

In Section 3 we mention that the determination of the entry point is part of the vulnerability discovery process. Therefore, BOPC assumes that the entry point is given. Without having access to actual exploits (or crashes), inference of entry points (and bugs) is undetermined. Hence, we have selected arbitrary locations as the entry points. This allows BOPC to find payloads for the evaluation without having access to *concrete* exploits. In practice, BOPC would leverage the given entry points as starting points. We demonstrate several test cases where the entry points are precisely at the start of functions, deep in the Call Graph, to show the power of our approach. Orthogonally, we allow for vulnerabilities to exist in the middle of a function. In such situations, BOPC would set our entry point to the location *after* the return of the function.

The lack of the exact entry point complicates the verification of our solutions. We leverage a debugger to "simulate" the AWP and modify the memory on the fly, as we reach the given entry point. We ensure as we step through our trace that we maintain the properties of the SPL payload expressed. That is, blocks between the statements are non-clobbering in terms of register allocation and memory assignment.

7 CASE STUDY: NGINX

We utilize a version of the nginx web server with a known memory corruption vulnerability [9] that has been exploited in the wild to further study BOPC. When an HTTP header contains the "Transfer-Encoding: chunked" attribute, nginx fails to properly length check for the received packet chunks, resulting in stack buffer overflow. This buffer overflow [16] results in an arbitrary memory write, fulfilling the AWP requirement. For our case study we select three of the most interesting payloads: spawning a shell, an infinite loop, and a conditional branch. Table 7 shows metrics collected during the BOPC execution for these cases.

7.1 Spawning a shell

Function `ngx_execute_proc` is invoked through a function pointer, with the second argument (passed to `rsi`, according to x64 calling convention), being a **void** pointer that is interpreted as a `struct` to initialize all arguments of `execve`:

```
mov   rbx, rsi
mov   rdx, QWORD PTR [rsi+0x18]
mov   rsi, QWORD PTR [rsi+0x10]
mov   rdi, QWORD PTR [rbx]
call  0x402500 <execve@plt>
```

BOPC leverages this function to successfully synthesize the `execve` payload (shown on the right side of Table 1) and generate a PoC exploit in less than a minute as shown in Table 7.

Assuming that `rsi` points to some writable address x, BOPC produces the following $(address, value, size)$ tuples: $(\$y, \$x, 8)$, $(\$y + 8h, 0, 8)$, $(\$x, /bin/sh, 8)$, $(\$x + 10h, \$y, 8)$, $(\$x + 18h, 0, 8)$, were $\$y$ is a concrete writable addresses set by BOPC.

Program	SPL payload												
	regset4	*regref4*	*regset5*	*regref5*	*regmod*	*memrd*	*memwr*	*print*	*execve*	*abloop*	*infloop*	*ifelse*	*loop*
ProFTPd	✓	✓	✓	✓	✓	✓	✓	✓ 32	✗$_1$	✓ 128+	✓ ∞	✓	✓ 3
nginx	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✓	✓ 128+	✓ ∞	✓	✓ 128
sudo	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✓ 128+	✗$_4$	✗$_4$
orzhttpd	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_1$	✗$_4$	✓ 128+	✗$_4$	✗$_3$
wuftdp	✓	✓	✓	✓	✓	✓	✓	✓	✗$_1$	✓ 128+	✓ 128+	✗$_4$	✗$_3$
nullhttpd	✓	✓	✓	✓	✓	✓	✗$_3$	✗$_3$	✓	✓ 30	✓ ∞	✗$_4$	✗$_3$
opensshd	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_4$	✗$_4$	✓ 512	✓ 128+	✓	✓ 99
wireshark	✓	✓	✓	✓	✓	✓	✓	✓ 4	✗$_1$	✓ 128+	✓ 7	✓	✓ 8
apache	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_4$	✓ ∞	✓ 128+	✓	✗$_4$
smbclient	✓	✓	✓	✓	✓	✓	✓	✓ 1	✗$_1$	✓ 1057	✓ 128+	✓	✓ 256

Table 6: Feasibility of executing various SPL payloads for each of the vulnerable applications. An ✓ means that the SPL payload was successfully executed on the target binary while a ✗ indicates a failure, with the subscript denoting the type of failure (✗$_1$ = Not enough candidate blocks, ✗$_2$ = No valid register/variable mappings, ✗$_3$ = No valid paths between functional blocks and ✗$_4$ = Un-satisfiable constraints or solver timeout). Note that in the first two cases (✗$_1$ and ✗$_2$), we know that there is *no* solution while, in the last two (✗$_3$ and ✗$_4$), a solution might exists, but BOPC cannot find it, either due to over-approximation or timeouts. The numbers next to the ✓ in *abloop*, *infloop*, and *loop* columns indicate the maximum number of iterations. The number next to the *print* column indicates the number of character successfully printed to the stdout.

Payload	Time	$\|C_B\|$	Mappings	$\|\delta G\|$	$\|H_k\|$
execve	0m:55s	10,407	142,355	1	1
infloop	4m:45s	9,909	14	1	1
ifelse	1m:47s	10,782	182	4	2

Table 7: Performance metrics for BOPC on nginx. *Time* **= time to synthesize exploit,** $|C_B|$ **= # candidate blocks,** *Mappings* **= # concrete register and variable mappings,** $|\delta G|$ **= # delta graphs created,** $|H_k|$ **= # of induced subgraphs tried.**

7.2 Infinite loop

Here we present a payload that generates a trace that executes an infinite loop. The *infloop* payload, is an simple infinite loop that consists of only two statements:

```
void payload() {
  LOOP:
    __r1 = 0;
    goto LOOP;
}
```

We set the entry point at the beginning of `ngx_signal_handler` function which is a signal handler that is invoked through a function pointer. Hence, this point is reachable through control-flow hijacking. The solution synthesized by BOPC is shown in Figure 6. The box on the top-left corner demonstrates how the memory is initialized to satisfy the constraints.

Virtual register `__r0` was mapped to hardware register `r14`, so `ngx_signal_handler` contains three candidate blocks, marked as octagons. Exactly one of them is selected to be the functional block while the others are avoided by the dispatcher blocks. The dispatcher finds a path from the entry point to the first functional block and then, finds a loop to return back to the same functional block (highlighted with blue arrows). Note that the size of the dispatcher block exceeds 20 basic blocks, while the functional block consists of a single basic block.

The oval nodes in Figure 6 indicate basic blocks that are outside of the current function. At basic block `0x41C79F`, function `ngx_time_sigsafe_update` is invoked. Due to the shortest path heuristic, BOPC, tries to execute as few basic blocks as possible from this function. In order to do so BOPC sets `ngx_time_lock` a non-zero value, thus causing this function to return quickly. BOPC successfully synthesizes this payload in less than 5 minutes.

7.3 Conditional statements

This case study shows an SPL if-else condition that implements a logical NOT. That is, if register `__r0` is zero, the payload sets `__r1` to one, otherwise `__r1` becomes zero. The execution trace starts at the beginning of `ngx_cache_manager_process_cycle`. This function is called through a function pointer. A part of the CFG starting from this function is shown in Appendix D. After trying 4 mappings, `__r0` and `__r1` map to `rsi` and `r15` respectively. The resulting delta graph is the shown in Figure 7.

As we mentioned in Section 5.6, when BOPC encounters a functional block for a conditional statement, it *clones* the current state of the symbolic execution and the two clones independently continue the execution. The constraints up to the conditional jump are the following:

```
0x41eb23 : $rdi = ngx_cycle_t* cycle
0x40f709 : *(ngx_event_flags + 1) == 0x2
0x41dfe3 : __r0 = rsi = 0x0
0x403cdb : $r15 = 0x1
           ngx_module_t ngx_core_module.index = 0
           $alloca_1 = *cycle
           ngx_core_conf_t* conf_ctx =
              *$alloca_1 + ngx_core_module.index * 8
0x403d06 : test rsi, rsi (__r0 != 0)
0x403d09 : jne  0x403d1b <ngx_set_environment+64>
```

If the condition is false and the jump is *not* taken, the following constraints are also added to the state.

```
0x403d0b : conf_ctx->environment != 0
0x403fd9 : __r1 = *($stack - 0x178) = 1;
```

When the condition is true, the execution trace will follow the "taken" branch of the trace. In this case the shortest path to the next functional block is $403d1b \rightarrow 403d3d \rightarrow 403d4b \rightarrow 403d54 \rightarrow 403d5a \rightarrow 403fb4$ with a total length 6. Unfortunately, this cannot be used as a dispatcher block, due to an exception that is raised at $403d4b$. The register `rsi`, is 1 and therefore when we attempt

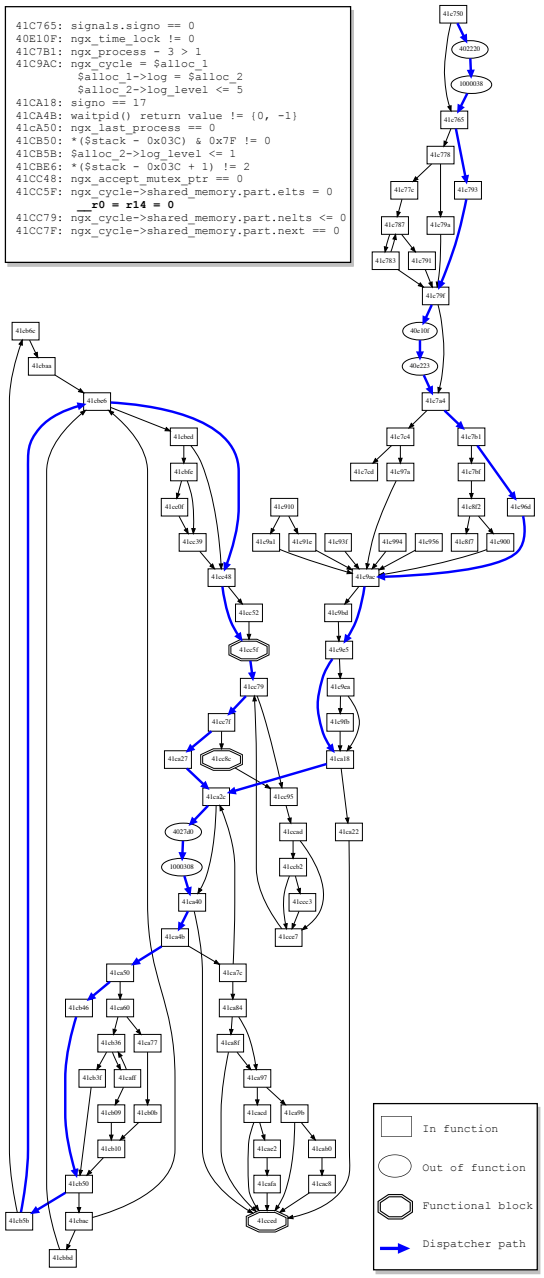

Figure 6: CFG of nginx's `ngx_signal_handler` and payload for an infinite loop (blue arrow dispatcher blocks, octagons functional blocks) with the entry point at the function start. The top box shows the memory layout initialization for this loop. This graph was created by BOPC.

to execute the following instruction: `cmp BYTE PTR [rsi], 54h`, we essentially try to dereference address 1. BOPC is aware of this exception, so it discards the current path and tries with the second shortest path. The second shortest path has length 7 and avoids the problematic block: $403d1b \rightarrow 403d8b \rightarrow 4050ba \rightarrow 40511c \rightarrow 40513a \rightarrow 403d9c \rightarrow 403da5 \rightarrow 403fb4$. This results in a new set of constraints as shown below:

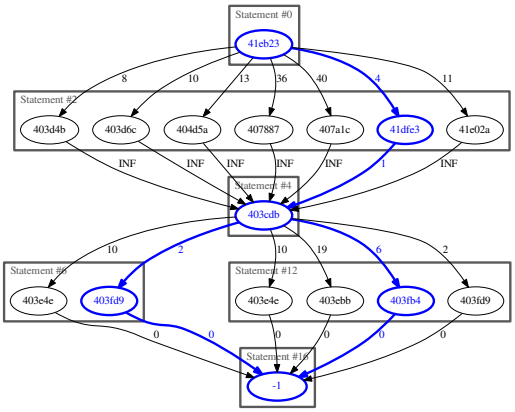

Figure 7: A delta graph instance for an *ifelse* payload for nginx. The first node is the *entry point*. Blue nodes and edges form the *minimum induced subgraph*, H_k. Statement #4 is conditional, execution branches into two statements. Note that BOPC created this graph.

```
0x403d1b : conf_ctx->env.elts = &elt (ngx_array_t*)
           conf_ctx->env.nelts == 0
0x4050ba : conf_ctx->env.nelts != $alloca_2->env.nalloc
0x40511c : conf_ctx->env.nelts += 1
0x40513a : $ret = conf_ctx->env.elts +
                  conf_ctx->env.nelts*conf_ctx->env.size
0x403d9c : $ret != 0
0x403da5 : conf_ctx->env.nelts != 0
0x403fb4 : __r1 = r15 = 0
```

8 DISCUSSION AND FUTURE WORK

Our prototype demonstrates the feasibility and scalability of automatic construction of BOP chains through a high level language. However, we note some potential optimizations that we consider for future versions of BOPC.

BOPC is limited by the *granularity* of basic blocks. That is, a combination of basic blocks could potentially lead to the execution of a desired SPL statement, while individual blocks might not. Take for instance an instruction that sets a virtual register to 1. Assume that a basic block initializes `rcx` to 0, while the following block increments it by 1; a pattern commonly encountered in loops. Although there is no functional block that directly sets `rcx` to 1, the combination of the previous two has the desired effect. BOPC can be expanded to address this issue if the basic blocks are coalesced into larger blocks that result in a new CFG.

BOPC sets several upper bounds defined by user inputs. These configurable bounds include the upper limit of (i) SPL payload permutations (P), (ii) length of continuous blocks (L), (iii) of minimum induced subgraphs extracted from the delta graph (N), and (iv) dispatcher paths between a pair of functional blocks (K). These upper bounds along with the timeout for symbolic execution, reduce the search space, but prune some potentially valid solutions. The evaluation of higher limits may result to alternate or more solutions being found by BOPC.

9 CONCLUSION

Despite the deployment of strong control-flow hijack defenses such as CFI or shadow stacks, data-only code reuse attacks remain possible. So far, configuring these attacks relies on complex manual analysis to satisfy restrictive constraints for execution paths.

Our BOPC mechanism automates the analysis of the remaining attack surface and synthesis of exploit payloads. To abstract complexity from target programs and architectures, the payload is expressed in a high-level language. Our novel code reuse technique, *Block Oriented Programming*, maps statements of the payload to functional basic blocks. Functional blocks are stitched together through dispatcher blocks that satisfy the program CFG and avoid clobbering functional blocks. To find a solution for this NP-hard problem, we develop heuristics to prune the search space and to evaluate the most probable paths first.

The evaluation demonstrates that the majority of 13 payloads, ranging from typical exploit payloads to loops and conditionals are successfully mapped 81% of the time across 10 programs. Upon acceptance, we will release the source code of our proof of concept prototype along with all of our evaluation results.

REFERENCES

[1] CVE-2000-0573: Format string vulnerability in wu-ftpd 2.6.0. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0573, 2001.

[2] CVE-2001-0144: Integer overflow in openssh 1.2.27. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144, 2001.

[3] CVE-2002-1496: Heap-based buffer overflow in null http server 0.5.0. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1496, 2004.

[4] CVE-2006-3747: Off-by-one error in apache 1.3.34. Available from MITRE, CVE-ID CVE-2006-3747, 2006.

[5] CVE-2006-5815: Stack buffer overflow in proftpd 1.3.0. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5815, 2006.

[6] CVE-2009-1886: Format string vulnerability in smbclient 3.2.12. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886, 2009.

[7] Orzhttpd - format string. https://www.exploit-db.com/exploits/10282/, 2009.

[8] CVE-2012-0809: Format string vulnerability in sudo 1.8.3. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0809, 2012.

[9] CVE-2013-2028: Nginx http server chunked encoding buffer overflow 1.4.0. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028, 2013.

[10] CVE-2014-2299: Buffer overflow in wireshark 1.8.0. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2299, 2014.

[11] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* (2009).

[12] Avgerinos, T., Cha, S. K., Rebert, A., Schwartz, E. J., Woo, M., and Brumley, D. Automatic exploit generation. *Communications of the ACM 57*, 2 (2014), 74–84.

[13] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011).

[14] Burow, N., Carr, S. A., Brunthaler, S., Payer, M., Nash, J., Larsen, P., and Franz, M. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* (2018).

[15] Cadar, C., Dunbar, D., Engler, D. R., et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008).

[16] Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security* (2015).

[17] Carlini, N., and Wagner, D. ROP is still dangerous: Breaking modern defenses. In *USENIX Security* (2014).

[18] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006).

[19] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010).

[20] Cheng, Y., Zhou, Z., Miao, Y., Ding, X., DENG, H., et al. ROPecker: A generic and practical approach for defending against ROP attack.

[21] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*. The MIT press, 2009.

[22] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security* (1998).

[23] Dang, T. H., Maniatis, P., and Wagner, D. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), ACM, pp. 555–566.

[24] Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security* (2014).

[25] Davi, L., Sadeghi, A.-R., and Winandy, M. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011).

[26] Designer, S. return-to-libc attack. *Bugtraq, Aug* (1997).

[27] Ding, R., Qian, C., Song, C., Harris, B., Kim, T., and Lee, W. Efficient protection of path-sensitive control security.

[28] Durden, T. Bypassing PaX ASLR protection. *Phrack magazine #59* (2002).

[29] Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., and Sidiroglou-Douskos, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[30] Follner, A., Bartel, A., Peng, H., Chang, Y.-C., Ispoglou, K., Payer, M., and Bodden, E. PSHAPE: Automatically combining gadgets for arbitrary method execution. In *International Workshop on Security and Trust Management* (2016).

[31] Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014).

[32] Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., and Franz, M. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies* (2012), USENIX Association, pp. 7–7.

[33] Hu, H., Chua, Z. L., Adrian, S., Saxena, P., and Liang, Z. Automatic generation of data-oriented exploits. In *USENIX Security* (2015).

[34] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016).

[35] Jacobson, E. R., Bernat, A. R., Williams, W. R., and Miller, B. P. Detecting code reuse attacks with a model of conformant program execution. In *International Symposium on Engineering Secure Software and Systems* (2014).

[36] Kahn, A. B. Topological sorting of large networks. *Communications of the ACM* (1962).

[37] Katoch, V. Whitepaper on bypassing aslr/dep. Tech. rep., Secfence, Tech. Rep., September 2011.[Online]. Available: http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf.

[38] Kil3r, and Bulba. Bypassing stackguard and stackshield. *Phrack magazine #53* (2000).

[39] King, J. C. Symbolic execution and program testing. *Communications of the ACM* (1976).

[40] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *OSDI* (2014), vol. 14, p. 00000.

[41] Microsoft. Visual studio 2015 — compiler options — enable control flow guard, 2015. https://msdn.microsoft.com/en-us/library/dn919635.aspx.

[42] Müller, T. ASLR smack & laugh reference. *Seminar on Advanced Exploitation Techniques* (2008).

[43] Müller, U. Brainfuck–an eight-instruction turing-complete programming language. *Available at the Internet address http://en. wikipedia. org/wiki/Brainfuck* (1993).

[44] Niu, B., and Tan, G. Modular control-flow integrity. *ACM SIGPLAN Notices 49* (2014).

[45] Niu, B., and Tan, G. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[46] Pakt. ropc: A turing complete rop compiler. https://github.com/pakt/ropc, 2013.

[47] Pappas, V. kBouncer: Efficient and transparent rop mitigation. *tech. rep. Citeseer* (2012).

[48] PAX-TEAM. Pax aslr (address space layout randomization). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[49] Payer, M., Barresi, A., and Gross, T. R. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2015).

[50] Polychronakis, M., and Keromytis, A. D. ROP payload detection using speculative code execution. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on* (2011).

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

[52] Salwan, J., and Wirth, A. ROPGadget. https://github.com/JonathanSalwan/ROPgadget, 2012.

[53] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security and Privacy (SP), 2015 IEEE*

Symposium on (2015).

[54] Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: Exploit hardening made easy. In *USENIX Security Symposium* (2011).

[55] Sen, K., Marinov, D., and Agha, G. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes* (2005), vol. 30, ACM, pp. 263–272.

[56] Shacham, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of CCS 2007* (Oct. 2007), S. De Capitani di Vimercati and P. Syverson, Eds., ACM Press, pp. 552–61.

[57] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security* (2004).

[58] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al. SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016).

[59] Tang, J., and Team, T. M. T. S. Exploring control flow guard in windows 10. *Available at "http://blog.trendmicro.com/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10"* (2015).

[60] The Chromium Projects. Control Flow Integrity The Chromium Projects. "https://www.chromium.org/developers/testing/control-flow-integrity".

[61] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security* (2014).

[62] Uno, T. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. *Algorithms and Computation* (1997).

[63] van de Ven, A., and Molnar, I. Exec shield. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[64] van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., and Giuffrida, C. Practical Context-Sensitive CFI. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (October 2015).

[65] van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., and Giuffrida, C. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), pp. 1675–1689.

[66] Wojtczuk, R. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e* (2001).

[67] Yen, J. Y. Finding the k shortest loopless paths in a network. *management Science 17*, 11 (1971), 712–716.

A EXTENDED BACKUS-NAUR FORM OF SPL

$\langle SPL \rangle$ ::= **void payload( )** { $\langle stmts \rangle$ }
$\langle stmts \rangle$ ::= ($\langle stmt \rangle$ | $\langle label \rangle$)* $\langle return \rangle$?
$\langle stmt \rangle$ ::= $\langle varset \rangle$ | $\langle regset \rangle$ | $\langle regmod \rangle$ | $\langle call \rangle$
| $\langle memwr \rangle$ | $\langle memrd \rangle$ | $\langle cond \rangle$ | $\langle jump \rangle$

$\langle varset \rangle$ ::= **int64** $\langle var \rangle$ = $\langle rvalue \rangle$;
| **int64*** $\langle var \rangle$ = {$\langle rvalue \rangle$ (, $\langle rvalue \rangle$)*};
| **string** $\langle var \rangle$ = $\langle str \rangle$;
$\langle regset \rangle$ ::= $\langle reg \rangle$ = $\langle rvalue \rangle$;
$\langle regmod \rangle$ ::= $\langle reg \rangle$ $\langle op \rangle$= $\langle number \rangle$;
$\langle memwr \rangle$ ::= *$\langle reg \rangle$ = $\langle reg \rangle$;
$\langle memrd \rangle$ ::= $\langle reg \rangle$ = *$\langle reg \rangle$;
$\langle call \rangle$ ::= $\langle var \rangle$ **(** (ϵ | $\langle reg \rangle$ (, $\langle reg \rangle$)* **);**
$\langle label \rangle$ ::= $\langle var \rangle$**:**
$\langle cond \rangle$ ::= **if (**$\langle reg \rangle$ $\langle cmpop \rangle$ $\langle number \rangle$**) goto** $\langle var \rangle$;
$\langle jump \rangle$ ::= **goto** $\langle var \rangle$;
$\langle return \rangle$ ::= **returnto** $\langle number \rangle$;

$\langle reg \rangle$:= '___r'$\langle regid \rangle$
$\langle regid \rangle$:= [0-7]
$\langle var \rangle$:= [a-zA-Z_][a-zA-Z_0-9]*
$\langle number \rangle$:= ('+' | '−') [0-9]+ | '0x'[0-9a-fA-F]+
$\langle rvalue \rangle$:= $\langle number \rangle$ | '&' $\langle var \rangle$
$\langle str \rangle$:= [.]*
$\langle op \rangle$:= '+' | '−' | '*' | '/' | '&' | '|' | '~' | '<<' | '<<'
$\langle cmpop \rangle$:= '==' | '!=' | '>' | '>=' | '<' | '<='

B STITCHING BOP GADGETS IS NP-HARD

We present the NP-hardness proof for the BOP Gadget stitching problem. This problem reduces to the problem of finding the *minimum induced subgraph* H_k in a delta graph. Furthermore, we show that this problem cannot even be approximated.

Let δG be a *multipartite* directed weighted delta graph with k sets. Our goal is to select *exactly* one node (i.e., functional block) from each set and form the *induced subgraph* H_k, such that the total weight of all of edges is *minimized*:

$$\min_{H_k \subset \delta G} \sum_{e \in H_k} distance(e) \tag{2}$$

A δG is *flat*, when all edges from i^{th} set are towards $(i+1)^{th}$ set. The nodes and the black edges in Figure 8 are such an example. In this case, the minimum induced subgraph, is the minimum among all *shortest paths* that start from some node in the first set and end in any node in the last set. However, if the δG is *not* flat (i.e., the SPL payload contains jump statements, so edges from i^{th} set can go anywhere), the shortest path approach does not work any more. Going back in Figure 8, if we make some loops (add the blue edges), the previous approach does not give the correct solution.

It turns out that the problem is NP-hard if the δG is not flat . To prove this, we will use a reduction from *K-Clique*: First we apply some equivalent transformations to the problem. Instead of having K independent sets, we add an edge with ∞ weight between every

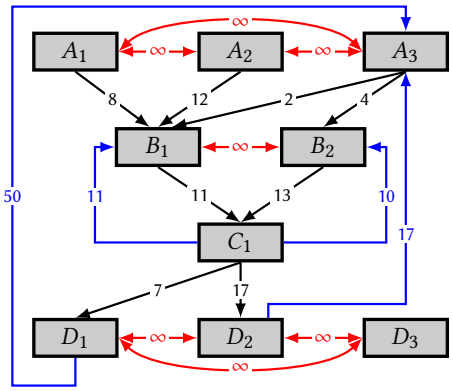

Figure 8: An delta graph instance. The nodes along the black edges form a *flat* delta graph. In this case, the *minimum induced subgraph*, H_k is A_3, B_1, C_1, D_1, with a total weight of 20, which is also the *shortest path* from A_3 to D_1. When delta graph is *not* flat (assume that we add the blue edges), the shortest path nodes constitute an induced subgraph with a total weight of 70. However H_k has total weight 34 and contains A_3, B_2, C_1, D_2. Finally, the problem of finding the minimum induced subgraph becomes equivalent to finding a k-clique if we add the red edges with ∞ cost between all nodes in the same set.

pair on the same set, as shown in Figure 8 (red edges). Then, the minimum weight K-induced subgraph H_k, cannot have two nodes from the same layer, as this would imply that H_k contains an edge with ∞ weight.

Let R be an undirected un-weighted graph that we want to check whether it has a k-clique. That is, we want to check whether $clique(R, k)$ is True or not. Thus, we create a new directed graph R' as follows:

- R' contains all the nodes from R
- $\forall$ edge $(u, v) \in R$, we add the edges (u, v) and (v, u) in R' with $weight = 0$
- $\forall$ edge $(u, v) \notin R$, we add the edges (u, v) and (v, u) in R' with $weight = \infty$

Then we try to find the *minimum weight k-induced subgraph* H_k in R'. It is true that:

$$\sum_{e \in H_k} weight(e) < \infty \Leftrightarrow clique(R, k) = True$$

$:\Rightarrow$ If the total edge weight of H_k is not ∞, this implies that for every pair of nodes in H_k, there is an edge with weight 1 in R' and thus an edge in R. This by definition means that the nodes of H_k form a k-clique in R. Otherwise (the total edge weight of H_k is ∞) it means that it does not exist a set of k nodes in R' that has all edge weights $< \infty$.

$:\Leftarrow$ If R has a k-clique, then there will be a set of k nodes that are fully connected. This set of nodes will have no edge with ∞ weight in R'. Thus, these nodes will form an induced subgraph of R' and the total weight will be smaller than ∞.

This completes the proof that finding the minimum induced subgraph in δG is NP-hard. However, no (multiplicative) approximation algorithm does exists, as it would also solve the K-Clique problem (it must return 0 if there is a K-Clique).

C SPL IS TURING-COMPLETE

We present a constructive proof of Turing-completeness through building an interpreter for Brainfuck [43], a Turing-complete language in the following listing. This interpreter is written using SPL with a Brainfuck program provided as input in the SPL payload.

```
1  int64 *tape  = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
2  string input = ".+[.+]";
3  __r0 = &tape;       // Data pointer
4  __r2 = &input;      // Instruction pointer
5  __r6 = 0;           // STDIN
6  __r7 = 1;           // STDOUT
7  __r8 = 1;           // Count arg for write/read
8  NEXT:    __r1 = *__r2;
9           if (__r1 != 0x3e) goto LESS;    // '>'
10          __r0 += 1;
11 LESS:    if (__r1 != 0x3c) goto PLUS;    // '<'
12          __r0 -= 1;
13 PLUS:    if (__r1 != 0x2b) goto MINUS;   // '+'
14          *__r0 += 1;
15 MINUS:   if (__r1 != 0x2d) goto DOT;     // '-'
16          *__r0 -= 1;
17 DOT:     if (__r1 != 0x2e) goto COMMA;   // '.'
18          write(__r7, __r0, __r8);
19 COMMA:   if (__r1 != 0x2c) goto OPEN;    // ','
20          read(__r6, *__r0, __r8);
21 OPEN:    if (__r1 != 0x5b) goto CLOSE;   // '['
22          if (__r0 != 0) goto CLOSE;
23          __r3 = 1;         // Loop depth counter
24 FIND_C:  if (__r3 <= 0) goto CLOSE;
25          __r2 += 1;
26          __r1 = *__r2;
27          if (__r1 != 0x5b) goto CHECK_C; // '['
28          __r3 += 1;
29 CHECK_C: if (__r1 != 0x5d) goto FIND_C;  // ']'
30          __r3 -= 1;
31          goto FIND_C;
32 CLOSE:   if (__r1 != 0x5d) goto END;     // ']'
33          if (__r0 != 0) goto END;
34          __r3 = 1;         // Loop depth counter
35 FIND_O:  if (__r3 <= 0) goto END;
36          __r2 -= 1;
37          __r1 = *__r2;
38          if (__r1 != 0x5b) goto CHECK_O; // '['
39         __r3 -= 1;
40 CHECK_O: if (__r1 != 0x5d) goto FIND_O;  // ']'
41        __r3 += 1;
42        goto FIND_O;
43 END:    __r2 += 1;
44 goto NEXT;
```

D CFG OF NGINX AFTER PRUNING

The following graph, is a portion of nginx's CFG that includes function calls starting from the function `ngx_cache_manager_process_cycle`. The graph only displays functions which are

up to 3 function calls deep to simplify visualization. Note the reduction in search space–which is a result of BOPC's pruning–as this portion of the CFG reduces to the small delta graph in Figure 7.