Zero-Overhead Path Prediction with Progressive Symbolic Execution

Richard Rutledge, Sunjae Park, Haider Khan, Alessandro Orso, Milos Prvulovic, and Alenka Zajic
Georgia Institute of Technology
Atlanta, USA
{rrutledge, sunjae.park, khan}@gatech.edu {orso, milos}@cc.gatech.edu {alenka.zajic}@ece.gatech.edu

Abstract—**In previous work, we introduced zero-overhead profiling (ZOP), a technique that leverages the electromagnetic emissions generated by the computer hardware to profile a program without instrumenting it. Although effective, ZOP has several shortcomings: it requires test inputs that achieve extensive code coverage for its training phase; it predicts path profiles instead of complete execution traces; and its predictions can suffer unrecoverable accuracy losses. In this paper, we present zero-overhead path prediction (ZOP-2), an approach that extends ZOP and addresses its limitations. First, ZOP-2 achieves high coverage during training through progressive symbolic execution (PSE)—symbolic execution of increasingly small program fragments. Second, ZOP-2 predicts complete execution traces, rather than path profiles. Finally, ZOP-2 mitigates the problem of path mispredictions by using a stateless approach that can recover from prediction errors. We evaluated our approach on a set of benchmarks with promising results; for the cases considered, (1) ZOP-2 achieved over 90% path prediction accuracy, and (2) PSE covered feasible paths missed by traditional symbolic execution, thus boosting ZOP-2's accuracy.**

Index Terms—**symbolic execution; path profiling; tracing**

I. Introduction

Program tracing consists of logging selected events during program execution. Such trace logs are then used for various tasks, such as computer forensics, debugging, performance analysis, and user profiling. Typically, program tracing is implemented through instrumentation, that is, by adding probes to a program that log events as they occur.

Albeit effective, instrumentation can cause issues due to its intrusive nature. In particular, instrumentation adds runtime overheads that can be problematic in many scenarios, including real-time systems, embedded software, and deployed applications. To address these issues, while still being able to collect accurate (partial) program traces, in previous work we developed zero-overhead profiling (ZOP) [1], a technique that can profile a program without instrumenting it by leveraging the electromagnetic (EM) emissions generated by the processing hardware during execution. ZOP, although effective, has three main limitations. *First*, it requires extensive code coverage, and therefore a thorough set of test inputs, during its training phase to achieve good accuracy. Basically, in ZOP, each relevant subpath must be executed, so that its EM signal can be recorded and later matched. Unfortunately, in real-world programs, the test cases are frequently few, of poor quality, and often completely absent. *Second*, ZOP predicts acyclic path profiles [2], rather than complete execution traces. These path profiles count executions of unique, acyclic paths within the program. Although useful for some tasks, path profiles summarize away the exact sequence of events that would instead be logged in a complete path trace. *Third*, ZOP can fail to recover from a misprediction. ZOP attempts to match EM signals by following the control flow graph of the program being profiled. When a misprediction occurs, ZOP backtracks until it can find a path that better matches the signal. Although this approach can avoid mispredictions that result in infeasible paths, the predicted and actual control flows can diverge beyond recovery when the EM signals collected during training do not closely match the signals observed during profiling for more than a short time.

In this paper, we propose zero-overhead path prediction (ZOP-2), a novel approach that extends ZOP and addresses its shortcomings. *First*, to support the training phase even in the absence of an extensive set of inputs, we developed a new input-generation technique based on symbolic execution: progressive symbolic execution (PSE). PSE overcomes some of the limitations of traditional symbolic execution by taking advantage of the fact that program subpaths need not be observed in the context of a complete execution. More precisely, initially PSE executes the whole program with symbolic inputs as done in classic symbolic execution. If this fails to achieve sufficient coverage for ZOP-2 training, it proceeds to execute functions with symbolic inputs and unconstrained global state (similar to UC-KLEE [3]). PSE then continues by (1) substituting called functions with symbolic stubs and (2) increasingly unconstrained local state, until a given coverage objective is achieved. Although this approach can result in infeasible paths, this is not problematic when the execution of such paths is used in training—in most (if not all) cases, the oversampled EM emissions will simply never match a signal produced during profiling.

Second, we modified the ZOP profiling phase to predict complete execution traces instead of acyclic-path profiles. In this way, we made the approach considerably more generable and applicable in a broader range of scenarios.

Finally, we developed a new signal matching algorithm that divides the EM signals into sampling windows and matches them in a stateless fashion (i.e., without following paths in the control flow graph of the program being profiled). Although this state-less prediction approach can increase the occurrence of mispredictions of individual basic blocks, these mispredictions do not impact future prediction accuracy, which means that ZOP-2 can easily recover from prediction errors.

To evaluate ZOP-2, we applied it to the three original ZOP benchmarks and to a new, larger benchmark. Our results show that ZOP-2 is a promising approach. In particular, they show that ZOP-2 does produce accurate path predictions, with an accuracy over 90% for the cases considered. They also show that PSE is an effective technique, in that it was able to cover feasible paths missed by traditional symbolic execution, which contributed to increasing ZOP-2's accuracy.

The main contributions of this paper are:

- ZOP-2, a zero-overhead whole-program tracing approach.
- PSE, an input generation technique that targets increasingly smaller code fragments until it achieves a given coverage goal.
- An implementation of PSE, which is publicly available as a docker image [4] and in archival format [5].
- An empirical evaluation that demonstrates the effectiveness and potential usefulness of ZOP-2.

II. Background

In this section, we provide some necessary background information on ZOP, our previous technique for zero-overhead (acyclic paths) profiling, and on symbolic execution. We also define some terms that we use in the rest of the paper.

A. Zero-Overhead Profiling

Zero-overhead profiling (ZOP) [1] computes acyclic path profiles [2] for a program P by observing the electromagnetic (EM) emanations produced by a computing system during execution of (an unmodified version of) P. ZOP consists of two main phases: training and profiling. In the training phase, ZOP runs P against a set of inputs to collect waveforms for the EM emanations generated by the computing system running P. In the profiling phase, ZOP (1) runs P, uninstrumented and unmodified, against inputs whose executions need to be profiled, (2) records the EM emissions produced by the program, and (3) matches these emissions with those collected during training to predict which acyclic paths were exercised and how often. In an evaluation performed on several benchmarks, ZOP was able to predict acyclic-path profiling information with an accuracy greater than 94% on average.

Despite these positive results, however, ZOP has some shortcomings that limit its usefulness and general applicability. First of all, as we discussed in the Introduction, ZOP requires an extensive set of inputs in order to build good models in the training phase. In fact, the empirical results we described above were obtained by using test suites that achieved complete branch coverage, which are rarely available in practice. In addition, acyclic path profiles provide useful information, but they summarize events into histograms and discard information about the full sequence of events. They therefore cannot be used for the many tasks for which information about complete traces is needed.

Finally, due to the way ZOP matches signals, the predictions it computes can suffer unrecoverable accuracy losses.

B. Symbolic Execution

Symbolic execution (SE) [6] is a technique that executes a program using symbolic instead of concrete inputs. At any point in the program's (symbolic) execution, SE keeps track of (1) the *symbolic state*, expressed as a function of the inputs, and (2) the *path condition (PC)*, a set of constraints in conjunctive form that consists of the conditions on the inputs under which the execution reaches that point. The symbolic state and the PC are built incrementally during SE. When SE executes a statement s that modifies the value of a memory location m, it computes the new symbolic value of m according to s's semantics and suitably updates the symbolic state. When SE executes a conditional branching statement c, it forks the execution, follows both branches, and updates the PC along each branch by adding an additional conjunct that represents c's predicate.

When successful, SE can compute an input that would cause a given point in the program to be reached. To do so, the PC for that point would be fed to an SMT (Satisfiability Modulo Theories) solver, which would try to find an assignment to the free variables in PC (i.e., the inputs) that satisfies the PC.

C. Terminology

A *control flow graph (CFG)* for a function f is a directed graph $G = \langle N, E, en, ex \rangle$, where N is a set of nodes that represent statements in f, $E \subseteq N \times N$ is a set of edges that represent the flow of control between nodes, and $en \in N$ and $ex \in N$ are the unique entry and exit points for the CFG.

A *basic block* in a CFG is a contiguous sequence of nodes (i.e., instructions) with no incoming branches except for the first node in the block and no outgoing branches except for the last node in the block.

A *call graph (CG)* is a directed graph $G =< M, E >$, where M is the set of functions in the program, and an edge $(f_a, f_b) \in E$ implies that function f_a may call function f_b.

III. Our Approach: Zero-Overhead Path Prediction

Figure 1 shows an overview of ZOP-2, our technique for zero-overhead path prediction. (Please note that, to avoid clutter, some elements in the figure are repeated.) As the figure shows, ZOP-2 consists of two main phases: Training and Prediction.

The *Training Phase* takes as input (the source code of) a *Program P*, whose complete paths we want to be able to predict, and generates the *EME Model*, a model of the electromagnetic (EM) emissions generated by the program. Two modules of ZOP-2 take part in this phase: the Input Generation and Replay module and the EME Model Generator. Given P, the goal of the *Input Generation and Replay* module is twofold. The first goal is to generate *Replay Cases*: inputs for P, or fragments thereof, that achieve a given coverage goal, typically expressed in terms of program subpaths. The second goal is to replay the generated inputs against the program (or against a program fragment), so that the *EME Model Generator* can record the EM emissions generated during

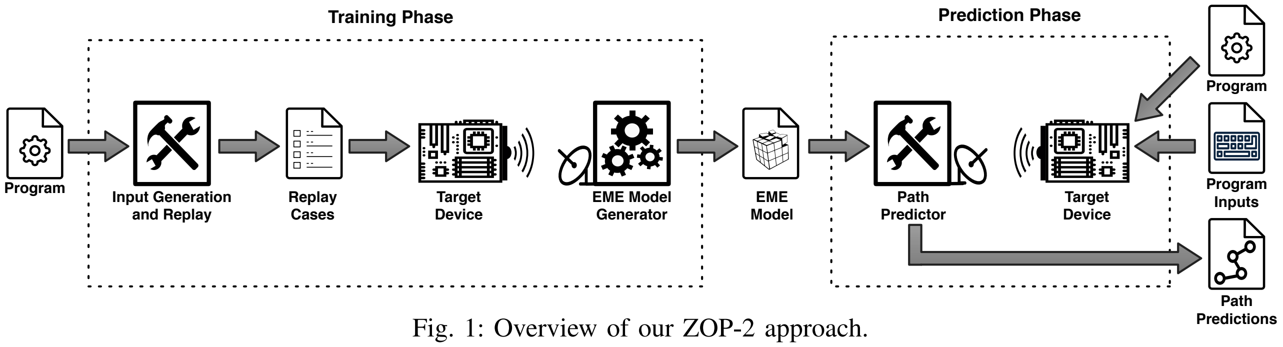

Fig. 1: Overview of our ZOP-2 approach.

the replay and generate the *EME Model*, which links such emissions to the part of the program that generated them.

The *Prediction Phase* takes as input the *EME Model*, *Program* P, and a set of *Program Inputs* for P, and generates a set of *Path Predictions*, one for each provided program input. The path predictions consist of complete execution traces for P and are computed by the *Path Predictor* module, which (1) observes the EM emanations produced by the *Target Device* as it runs P against the provided inputs and (2) matches the observed emanations with those in the *EME Model*.

In the rest of this section, we describe the four modules of ZOP-2 in detail.

A. Input Generation and Replay

The training phase of ZOP-2 requires the recording of sample EM emissions collected from the *Target Device* (i.e., the device that runs the program whose traces we want to collect). Moreover, for the training to be effective, ZOP-2 needs to collect a comprehensive set of samples. Ideally, the technique would need to collect one sample for every possible path in the program, which is clearly impractical. Because ZOP-2's signal matching divides the EM emissions for an execution into smaller sampling windows, however, having samples of subpaths whose length is comparable to that of the sampling window is typically enough. The goal of the Input Generation and Replay module is therefore to generate inputs that adequately cover a suitably identified set of subpaths within the program.

The first step performed by this module is a preprocessing of the source code that performs a set of semantics-preserving transformations aimed to facilitate later source code manipulation. Specifically, the preprocessing expands macros, refactors short-circuiting boolean expressions, encloses single statement blocks in braces, and rewrites each return statement that involves a complex value as an assignment to a temporary variable followed by a simple return of that variable.

Next, the module instruments the source code by inserting *markers* (i.e., special probes) at selected program points. These markers partition an execution trace into segments, which we refer to as *m2m paths* (marker-to-marker paths). These m2m paths are the subpaths within the program functions that we want the inputs to cover; that is, they are the coverage requirements for the input generation. The level of granularity of the inserted markers is critical to the effectiveness of the approach. Subpaths that are too short would be easy to cover but would result in EM signals that are hard to recognize and match. Conversely, subpaths that are too long would generate EM signals that are easy to recognize and match but would be difficult to cover. Based on our past experience [1], preliminary experimentation, the way ZOP-2 performs signal matching, and our domain knowledge, we selected the following points for inserting markers: entry nodes of functions, exit nodes of functions, loop heads, and target nodes of goto statements. Furthermore, because these markers can occasionally result in excessively long m2m paths, our techniques splits m2m paths longer than a selected threshold by suitably inserting additional markers. We selected these marking criterion so that (1) m2m paths are intra-procedural and do not contain cycles, (2) any program trace can be represented as a sequence of contiguous m2m paths, and (3) the length of the signals generated by the m2m paths is comparable to that of the sampling window used by the Path Predictor module (see Section III-C).

Given a complete set of m2m paths, our technique tries to generate inputs that cover all such paths using an approach based on symbolic execution. In principle, traditional symbolic execution can generate inputs for all feasible paths in a program. However, its effectiveness and scalability are limited in practice by several issues, and in particular by the path explosion problem—the fact that the number of feasible paths is usually exponential in the number of code branches [7]. In fact, traditional symbolic execution has problems covering even individual statements, let alone m2m paths, that are located deep in the call graph or hidden behind complex, looping control flow. To address this problem, and be able to generate inputs that cover most m2m paths, we defined a new technique that extends classical symbolic execution and that we call *progressive symbolic execution (PSE).*

1) Progressive Symbolic Execution (PSE): The key insight behind PSE is that a given m2m path mp in program P need not be observed along a complete path, that is, a path that starts from P's entry and follows a complete, actual execution. If PSE cannot generate an input that executes mp, it therefore derives a related program P' that contains an equivalent subpath for which it can find an input. To generate P', PSE operates along two dimensions: it (1) considers increasingly smallers fragments of the program and (2) replaces calls to other functions or libraries with symbolic stubs.

```
1   int P(char *p) {
2     mark(m1);
3     int x=complex(p);
4     while (x > 4) {
5       mark(m2);
6       // some code
7     }
8     // some code
9     mark(m3);
10  }
```

Listing 1: Original code.

```
1   int P′(char *p) {
2     mark(m1);
3     int x=symbolic_int();
4     while (x > 4) {
5       mark(m2);
6       // some code
7     }
8     // some code
9     mark(m3);
10  }
```

Listing 2: Modified code.

We illustrate this second dimension with the example shown in Listing 1. Assume that we are interested in covering m2m path $\langle 5, 6, 7, 8, 9\rangle$, between markers $m2$ and $m3$, and that the symbolic execution of function `complex()` either results in a timeout or cannot be performed because the code of the function involves theories not supported by the underlying solver. In such a case, no input covering the path of interest would be produced. However, it would be straightforward to generate an input that cover the analogous subpath $\langle 5, 6, 7, 8, 9\rangle$ in the derived program P' in Listing 2, where the integer value returned by `complex()` has been replaced with a symbolic integer.

PSE produces a set of inputs encoded as *Replay Cases*, where each replay case is an ordered pair that consists of a program fragment and inputs for that fragment. It also constructs, for each replay case, the scaffolding necessary to run the corresponding code fragment against its input.

The next two sections discuss how PSE (1) generates inputs that aim to cover all m2m paths identified in a program P and (2) runs P or fractions thereof against the generated inputs to support ZOP-2's training phase.

2) PSE: Input Generation: To try to reach all the m2m paths in a program, PSE progressively unconstrains program state in four phases or strategies. (The phases are progressive in that the symbolic state in each phase is a superset of the symbolic state in the prior phase.)

uInp (symbolic input state): Execution of the whole program with symbolic inputs. This is equivalent to traditional symbolic execution performed from the entry point of the program.

uExt (symbolic external state): Execution of each function with symbolic input parameters and symbolic global state. Intuitively, this strategy corresponds to executing a function as if it could be reached with every possible state and is equivalent to under-constrained symbolic execution [8].

uStub (symbolic stubs): Execution of each function with symbolic input parameters, symbolic global state, and all callees replaced by symbolic stubs (see example in Listings 1 and 2). Symbolic stubs return an unconstrained value and unconstrain global state and values passed as output parameters. Intuitively, in addition to executing a function as if it could be reached with every possible state, this strategy also assumes that callees can modify the state in every possible way.

uInt (symbolic internal state): Executions of fragments of a function with symbolic local and global state. Intuitively, this strategy corresponds to unconstraining the state reachable by a code fragment, so that the code fragment can be executed as if it could be reached with every possible state.

The specifics of PSE's input generation are described in Algorithm 1. The algorithm takes as input (1) a program, (2) a set of entry points for the program, and (3) three tuning timeout parameters, and produces as output a set of pairs of program fragments and corresponding inputs. (Although in most cases programs have a single entry point, supporting multiple entry points allows for applying the same approach to libraries.) The timeout values specify maximum time budgets for the progressive unconstraining strategies: $t0$ applies to uInp, $t1$ applies to uExt, and $t2$ applies to both uStub and uInt. The output of the algorithm consists of all the replay cases generated by all progressive phases, which are stored in the container initialized in line 2.

The algorithm starts by performing the uInp strategy on each of the program entry points (lines 4-5). The uExt strategy iterates over each program function as discovered in a breadth-first traversal of the program call graph and maintained in a worklist (lines 7-12). We selected this traversal order to lengthen the average replay trace, as entry from functions closer to program entry should produce longer traces. Lines 13-14 ensure that the algorithm only executes uExt if a remaining m2m path is reachable from the current function in the call graph traversal. In that case, the algorithm invokes PSE on the function with the uExt strategy, adds the resulting replay cases to set *cases*, and updates the set of remaining m2m paths (lines 15-17).

Note that, for clarity, we treat the utility function *m2mPaths* as polymorphic; that is, the function always returns a set of intra-procedural m2m paths contained within its single argument. If the argument is a function, it simply returns the set of m2m paths in the function. Conversely, if the argument is a program or a set of functions, it returns the union of the m2m paths in each function. Finally, if the argument is a basic block, it returns the set of m2m paths reachable from the block and within the function containing the block.

If there are remaining m2m paths within the current function, the algorithm aggressively unconstrains additional program state by substituting symbolic stubs for all callees within the function (uStub strategy). This strategy allows the algorithm to skip over complex callees that may be problematic for symbolic execution. (Since each function entry and exit is marked, m2m paths within the callee can be covered separately from the current function context.) Each replay case produced in this phase must be considered for retention (lines 21-27). These includes replay cases that result in a memory fault due to the increased amount of symbolic state. We retain these faulting replay cases anyway in case they end up being the only cases covering a specific m2m path.

When symbolic stubs fail to expose a remaining m2m path, the algorithm proceeds to the uInt phase, which unconstrains also the program state at specific points within a function (lines 28-39). This portion of the algorithm is analogous to the uStub strategy, except that it traverses the basic blocks

Algorithm 1: Input generation

```
Input  : $program$: program to analyze
         $entry\_points$: program entry points
         $t0$: classic symbolic execution timeout
         $t1$: PSE native callees timeout
         $t2$: PSE stubbed callees timeout
Output: $result : \{< frag,\ input >\}$
1  begin
2      $cases \leftarrow \emptyset$
3      $remaining \leftarrow$ m2mPaths(program)
4      foreach $fn \in entry\_points$ do
5          $cases \leftarrow cases \cup$ execPSE (program, fn, uInp, t0)
6      $remaining \leftarrow remaining \setminus$ coverage(cases)
7      $visited \leftarrow \emptyset$
8      $fn\_worklist \leftarrow entry\_points$
9      foreach $fn \in fn\_worklist$ do
10         $visited \leftarrow visited \cup \{fn\}$
11         $new\_fns \leftarrow$ callees(fn) $\setminus visited$
12         append(fn_worklist, new_fns)
13         $reachable\_fns \leftarrow$ reaching(fn)
14         if $remaining \cap$ m2mPaths(reachable_fns) $\neq \emptyset$ then
15             $new\_cases \leftarrow$ execPSE (program, fn, uExt, t1)
16             $cases \leftarrow cases \cup new\_cases$
17             $remaining \leftarrow remaining \setminus$ coverage(new_cases)
18             if $remaining \cap$ m2mPaths(fn) $\neq \emptyset$ then
19                 $faulting \leftarrow \emptyset$
20                 $new\_cases \leftarrow$ execPSE (program, fn, uStub, t2)
21                 foreach $case \in new\_cases$ do
22                     if $remaining \cap$ coverage(case) $\neq \emptyset$ then
23                         if faulted(case) then
24                             $faulting \leftarrow faulting \cup \{case\}$
25                         else if completed(case) then
26                             $cases \leftarrow cases \cup \{case\}$
27                             $remaining \leftarrow remaining \setminus$ coverage(case)
28                 if $remaining \cap$ m2mPaths(fn) $\neq \emptyset$ then
29                     $bb\_worklist \leftarrow \{bb \mid bb \in CFG(fn)\ sorted\ by\ BFS\}$
30                     foreach $bb \in bb\_worklist$ do
31                         if $remaining \cap$ m2mPaths(bb) $\neq \emptyset$ then
32                             $new\_cases \leftarrow$ execPSE (program, bb, uInt, t2)
33                             foreach $case \in new\_cases$ do
34                                 if $remaining \cap$ coverage(case) $\neq \emptyset$ then
35                                     if faulted(case) then
36                                         $faulting \leftarrow faulting \cup \{case\}$
37                                     else if completed(case) then
38                                         $cases \leftarrow cases \cup \{case\}$
39                                         $remaining \leftarrow remaining \setminus$ coverage(case)
40                     foreach $case \in faulting$ do
41                         if $remaining \cap$ coverage(case) $\neq \emptyset$ then
42                             $cases \leftarrow cases \cup \{case\}$
43                             $remaining \leftarrow remaining \setminus$ coverage(case)
44     $result \leftarrow \emptyset$
45     $remaining \leftarrow$ m2mPaths(program)
46     $cs\_worklist \leftarrow \{case \mid case \in cases\ sorted\ by\ trace\ length\}$
47     foreach $case \in cs\_worklist$ do
48         if $remaining \cap$ coverage(case) $\neq \emptyset$ then
49             $result \leftarrow result \cup \{case\}$
50             $remaining \leftarrow remaining \setminus$ coverage(case)
51     return $result$
```

within a functions's CFG instead of the functions within the call graph of the program. Also in this case, some replay cases may result in a memory fault and are retained in case they cover m2m path not otherwise covered.

Before advancing to the next function in the call graph traversal, the faulting replay cases are examined for coverage of this function's remaining m2m paths to decide which ones to keep (lines 39-43). Because faulting replay cases contain a record of the faulting basic block and the number of times that block occurs in the replay trace before faulting, PSE's replay can use these replay cases, if needed, and terminate them prior to the execution of the faulty statement.

In its final part, the algorithm returns a set of replay cases selected from all the potential replay cases generated. Longer replay traces preserve more of the processor context, and thus can generate more authentic EM emissions during training. Therefore, given a set of replay cases covering a given m2m path, the algorithm favors the case with the longest trace. It does so by sorting the candidate replay cases by trace length and greedily selecting cases to achieve maximum m2m path coverage (lines 44-50).

Algorithm 1 relies on function *execPSE* to perform the different phases of its progressive symbolic execution. We provide the details of *execPSE* in Algorithm 2. The inputs of the *execPSE* algorithm are (1) the program to symbolically execute, (2) the program point to be used as the starting point for the symbolic execution, (3) the strategy to be used, and (4) the timeout to be enforced.

The algorithm first sets*s* to the initial symbolic state, sets the first instruction to the first instruction in the *start* basic block, and initializes the set of active states with a the single element *s* (lines 2-4). It then unconstrains the formal parameters to the function containing the *start* basic block (lines 5-6).

The algorithm continues to unconstrain program state according to the specified strategy (lines 6-12). The instruction processing loop (lines 13-33) is the same used in traditional symbolic execution, except for the way it handles call instructions (lines 18-33). If either the callee*f* is an external function, or the unconstraining strategy is uStub or uInt, the algorithm (1) creates a new symbolic variable for each formal output parameter of *f* and (2) assigns this symbolic variable to the corresponding actual argument at the call site. Additionally, the algorithm creates new symbolic values and assignments for each global variable referenced by *f* and for *f*'s return value, if present.

3) PSE: Optimizations: To make PSE more scalable, we have incorporated in our technique several optimizations.

Lazy Initialization: PSE uses lazy initialization [9] to construct pointer inputs for execution at an arbitrary program point. Specifically, accessing an unconstrained pointer value causes PSE to explore potential program paths in which the pointer (a) is `null`, (b) points to a newly allocated memory object of the targeted type, or (c) points to an existing memory object of the targeted type. To prevent lazily initialized pointers to lazily initialized pointers from unrolling infinitely, PSE tracks the depth of lazy memory objects. When the depth exceeds a configurable threshold, only states for cases (a) and (c) above are considered.

Pointer Type-Casting: Type-casting between pointer types is a common practice in C programs. For example, a pointer may be declared as a `char *`, accessed, and later cast to `struct foo *`. In these cases, the lazily initialized memory behind the pointer may no longer be large enough to store memory objects of the new type. To address this issue, lazily initialized symbolic objects have an immutable

Algorithm 2: execPSE (simplified)

```
Input  : $program$: program to symbolically execute
         $start$: program point to begin PSE
         $strategy$: $uInp \mid uExt \mid uStub \mid uInt$
         $timeout$: maximum time to perform PSE
Output: $result : \{< frag,\ input >\}$
begin
    $s \leftarrow initial\_state[start]$
    $fn \leftarrow$ containingFn(start)
    $active\_states \leftarrow \{s\}$
    foreach $arg \in$ formalArgs(fn) do
        unconstrain(s, arg)
    if $strategy \in \{uExt, uStub, uInt\}$ then
        foreach $var \in globalvariables$ do
            unconstrain(s, var)
        if $strategy = uInt$ then
            foreach $var \in$ localVars(fn) do
                unconstrain(s, var)
    $stop \leftarrow now() + timeout$
    while $active\_states \neq \emptyset \wedge now() < stop$ do
        $s \leftarrow$ selectState(active_states)
        $inst \leftarrow$ nextInstruction(s)
        switch $inst$ do
            case Call do
                $f \leftarrow$ targetFunction(inst)
                if $strategy \in \{uStub, uInt\} \vee f \notin program$ then
                    ¡ foreach $arg \in$ actualArgs(inst) do
                        if isOutputPointer(arg) then
                            unconstrain(s, value)
                            $arg \leftarrow value$
                    foreach $var \in globalvariables$ do
                        if isReferenced(f, var) then
                            unconstrain(var, value)
                            $arg \leftarrow var$
                    if returnType($f$) $\neq void$ then
                        unconstrain(s, value)
                        setReturn(s, inst, value)
                else
                    executeCall(s, $f$)
```

maximum physical size and a flexible visible size. A lazy object's initial visible size depends on the size of the allocated type. A subsequent cast to a larger type can increase the visible size, up to its physical size, whereas a cast to a smaller type does not decrease it. The visible size is used when reporting symbolic solutions and when enforcing the inbounds pointer assumption, which we discuss next.

Inbounds Pointer Assumption: Since out-of-bound pointer accesses can result in non-deterministic behavior, lazy initialization ensures that unconstrained pointers either point to allocated memory or are `null`. However, there are other ways to have a potentially out-of-bounds pointer, such as through array indexing and pointer arithmetic. In PSE, an indexed operation automatically inserts a path constraint requiring the resulting pointer to be within the target allocation block. This approach reduces the path search space while only eliminating undesirable paths. Faulting or non-deterministic paths have in fact low utility when used to generate training samples.

Path Explosion Mitigation: Rather than mitigating path explosion using search strategies, PSE tries to eliminate undesirable states early using multiple heuristics. The inbounds pointer assumption discussed above, for instance, eliminates many abnormally terminating paths. Loops are also a significant cause of path explosion, as symbolic conditions within a loop body can create, at each iteration, a number of new paths exponential in the number of branches. To mitigate this issue, PSE periodically samples the number of active path states in each loop body. If the number of states in a single loop body grows across the sample interval by more than a configurable threshold, those paths are randomly reduced by 90%.

4) PSE: Input Replay: The replay cases (i.e., inputs) generated by PSE for a program P using its uInp strategy can be run directly on P. This is not true, however, for the replay cases generated by PSE using its uExt, uStub, and uInt strategies, for which PSE must create suitable scaffolding. The reason is that these replay cases are generated by unconstraining program state, considering fragments of the program, and replacing called function with symbolic stubs.

PSE generates the needed replay scaffolding in the same language as P, and the scaffolding consists of four major parts: replay bodies, replay stubs, input data, and replay harnesses.

The *replay bodies* contain the source statements that comprise the m2m paths recorded for ZOP-2 training. Bodies for replay cases generated using the uExt strategy simply consist of the original function and corresponding callees. Bodies for replay cases generated using the uStub strategy also consist of the original source function, but they are linked against newly created replay stubs that return the right values and suitably set output parameters and global variables. In addition to this, bodies for replay cases generated using the uInt strategy must also be able to (1) start executing from an internal basic block *bb*, (2) initialize the local and visible global state at *bb*, and (3) exit at the right point in the execution (i.e., after visiting a termination basic block a specific number of times). To do so, PSE first creates a copy of the original function containing the fragment of interest and identifies the statement $stmt$ from which to start the execution. It then inserts a `goto` instruction at the beginning of the current body, so as to cause the execution to jump to the $stmt$. Finally, it inserts a call to a function that suitably initializes local and global state.

The *replay stubs* correspond to symbolic stubs and provide suitable values for returns, output parameters, and global variables. Furthermore, because each called function can be invoked multiple times, PSE creates ordered sets of values, so as to be able to produce the right values for the different invocations. When executing a replay case with symbolic stubs, PSE substitutes these replay stubs to the original called functions in the corresponding replay body.

The *input data* consist of static-initialized data structure arrays in the original source language produced from the generated input values in the replay cases. Note that the data input set for a replay case may not contain complete values for all the required input variables. For instance, a function may read the value of an input pointer but only write to the pointed memory block. In general, since the values not contained in the input set do not affect the execution, PSE can safely initialize the missing data items with some default value. Among the input data, pointer variables require special handling, as the address space during input generation is different from the address space during replay. To address this issue, PSE adds to the replay cases a map of the address space at the time the case was generated. This map includes the address and size of each memory object, the allocation type, and a list of the cast operator types applied to the memory object. PSE needs the entire address space because pointer values that are not in the generated input set cannot be assigned a default value as PSE does with fundamental types. Writing memory by dereferencing such a pointer would in fact likely result in an access violation if not in undefined behavior. Given this map, if the pointer value resolves into a memory object in the address space of the replay case, PSE calculates its offset and emits the replay pointer value as an offset into the statically-initialized data structure for the target object. Otherwise, it is given a default initializer based on its type.

Finally, the *replay harnesses* contain one harness for each replay body and fragment, and a driver that invokes each replay fragment harness in turn. The fragment harness iterates through each input data set for the fragment, initializing global variables, declaring and initializing fragment parameters, and initializing substituted stubs.

B. EME Model Generator

The goal of the EME (EM Emissions) Model Generator is to generate the EME Model, which relates EM emissions to the code in the program that generated them. To do so, this module collects and analyzes EM emissions while replaying the inputs (i.e., replay cases) generated by PSE.

Because the control flow path induced by these inputs is known, so is the sequence of markers executed while replaying the inputs. Moreover, the marker probes log the processor clock cycle, so our technique knows at which clock cycles each m2m path started and ended and can use the marker sequence and the corresponding timestamps to annotate the collected EM emissions. More precisely, the EME Model Generator is able to mark the collected emissions so as to identify the starting and the ending point of each m2m path.

This information is stored in the *EME Model*, which is then used in the prediction phase to compute which (complete) paths are being traversed by unknown executions.

C. Path Predictor

This module is a fundamental part of ZOP-2, as it is the component that actually produces path predictions using the EME Model generated during the training phase.

As unknown inputs are provided to the program P for which ZOP-2 built the EME Model, and P runs on the Target Device, the Path Predictor collects the EM emissions produced by the device, which represent the *test signal*. It then splits this test signal into small, non-overlapping, and fixed-length segments, referred to as *sampling windows*.

Similar to what happens for the length of the m2m paths, the size of the sampling window can considerably affect the accuracy of the path prediction algorithm. On the one hand, if the window size is too small, the signal matching has low reliability, especially in the presence of measurement noise. On the other hand, if the size is too large, a single window may contain multiple m2m paths and require more training samples to be effective. Because of these tradeoffs, we performed preliminary experimentation to identify a good combination of lengths for m2m paths and window sizes.

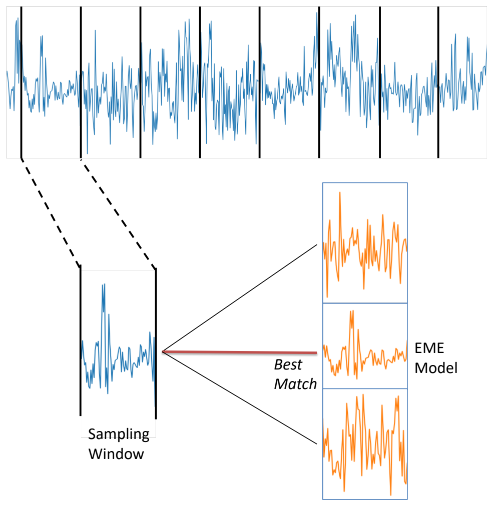

Fig. 2: EM emission matching approach.

Once the test signal is split into sampling windows, the Path Predictor searches, for each sampling window, the best match in the EME Model based on the least Euclidean distance [10], as intuitively shown in Figure 2. Our technique then uses the best matching sample in the EME Model, together with the annotations added by the EME Model Generator, to predict which m2m path contains the sampling window. A concatenation of all the so identified m2m paths is used to predict the entire control flow path corresponding to the observed, unknown execution.

In comparison to our original ZOP algorithm, which used a depth-first search (DFS) through the program's CFG while matching signals, this prediction mechanism matches each signal window independently (i.e., there is no "state" of the search). This approach is preferable to a DFS because this latter is prone to error propagation; any misprediction is often followed by a series of consecutive mispredictions, which amplifies the initial error and may result in the algorithm getting lost [1]. Furthermore, mispredictions in the original ZOP can also result in an exponential growth in the amount of backtracking the algorithm needs to perform, which can limit the scalability and applicability of the algorithm, especially in the case of large applications.

Because the stateless path prediction algorithm in ZOP-2 matches and predicts each window independently, it is free from error propagation. Moreover, unlike ZOP, ZOP-2's path prediction is embarrassingly parallel and could thus be easily distributed to increase its efficiency.

IV. Empirical Evaluation

To evaluate the effectiveness of our technique, we implemented it in a prototype tool and performed an empirical

TABLE I: Benchmark statistics.

Benchmark	LOC	Basic Blocks	M2M Paths
replace	495	245	229
schedule	464	175	153
print_tokens	579	178	153
mDNS	24,815	3,939	5,763

evaluation on a set of benchmarks. In our evaluation, we addressed the following research questions:

RQ1: Does ZOP-2 provide accurate path predictions?
RQ2: How does ZOP-2 compare to ZOP?
RQ3: To what extent does state unconstraining help coverage?

A. Implementation Details

We implemented the modules of ZOP-2 discussed in Section III. We used CIL [11] and clang [12] to preprocess source code. We used a combination of clang and NetworkX [13] for control flow graph analysis. We implemented PSE by extending the KLEE [14] symbolic execution engine. We also relied on LLVM [15] and on the STP constraint solver [16], Our implementation of PSE is publicly available as a self-contained docker image [4]. We implemented our path predictor module in Matlab 2018b [17].

B. Evaluation Setup

To answer our research questions we selected four benchmarks. The first three were used to evaluate ZOP in previous work [1]. The fourth benchmark, a real-world mDNS server, shows the scalability of our approach, as it is two orders of magnitude larger than the other benchmarks. Table I provides size metrics for our benchmarks. As the target device, we used an Altera Cyclone II FPGA with a Nios IIe processor. Using an FPGA lets us leverage various debugging features and I/O pins to better understand program behavior at the individual-cycle level. Unfortunately, however, it also considerably limits the size of the benchmarks we can consider.

Using the timeout parameters described in Section III-A2, we defined three variants of PSE. This allowed us, together with the use of vanilla KLEE, to evaluate the effects of different input generation techniques on ZOP-2's accuracy:

CS: Classic symbolic execution. Vanilla KLEE [18].
UC: Under-constrained symbolic execution. UC-KLEE proxied by performing PSE at the function level.
PG: PSE with all unconstraining strategies enabled.
FN: PSE without the uExt strategy. (Basically, this parameterization skips under-constrained symbolic execution by introducing symbolic stubs and considering sub-function fragments right away, which makes the analysis considerably faster at the cost of generating shorter paths.)

C. RQ1: Prediction Accuracy

To answer RQ1, we first generated replay cases (i.e., input sets) for the four benchmarks and for the four input generation strategies considered: CS, UC, FN, and PG. Second, we used ZOP-2 to train EME Models for each benchmark and input set. Then, for replace, schedule, and print_tokens, we randonly selected 100 inputs from the tests provided in the SIR repository [19]. For mDNS, we used Avahi [20] to generate 9 inputs (i.e., mDNS queries) that target different host addresses and ask for different services, including incorrect queries. (We manually checked that the inputs exercise a different aspect of the mDNS protocol.) Finally, we used ZOP-2 to perform path prediction using the generated EME Models.

TABLE II: Mean path prediction accuracy (%).

Benchmark	**CS**	**UC**	**FN**	**PG**
replace	90.05	93.89	90.31	93.89
schedule	94.27	94.31	94.45	94.44
print_tokens	77.71	93.35	78.72	93.36
mDNS	98.94	98.97	98.95	98.95

Table II reports the path prediction accuracy for the cases considered, computed by measuring the edit distance [21] between true and predicted paths. The prediction error is the edit distance over the length of the path, and the prediction accuracy is 1 minus the prediction error.

As the table shows, for all four benchmarks the prediction accuracy tends to be generally fairly high. The highest accuracy is achieved with either UC or PG in three of four cases, with the fourth case (schedule) showing a very close result for FN and PG. Interestingly, as we will show in section IV-E, PG achieves higher coverage but does not always result in higher prediction accuracy. The reason for this is that the increased accuracy from a fully populated waveform model is offset by an increased possibility of a misprediction when the sampling window straddles a the entry or exit of a symbolic stub. Approaches to match sub-window EM signals could address this issue and further improve the results for PG.

Also interestingly, ZOP-2 achieves high prediction accuracy for mDNS regardless of the input generation strategy involved.¡ Further analysis of the results showed that this happens for different reasons, with the main ones being that (1) all the paths the program takes when receiving a valid mDNS packet are similar because traces are largely dominated by loops with a large number of iterations, and (2) all the paths the program takes when receiving an invalid mDNS packet are extremely short. This skews the results considerably and makes it so that any input generation strategy that produces even just a few valid and invalid packets result in reliable EME Models, and thus high accuracy in the production.

To better understand how path prediction accuracy varies across inputs, consider the box-and-whisker plot in Figure 3, which shows detailed results for PG. (The plot for UC is fairly similar.) For each benchmark (x-axis), the figure shows the range of prediction accuracy (y-axis). The boxes represent the 1st and 3rd quartiles, with an interior band at the median. The whisker ends represent the lowest and highest points within 1.5 of the interquartile range. Dots signify outliers.

Although Figure 3 shows consistently high path prediction accuracy, it also shows that there is room for improvement. For example, accuracy above 96% for the schedule benchmark is

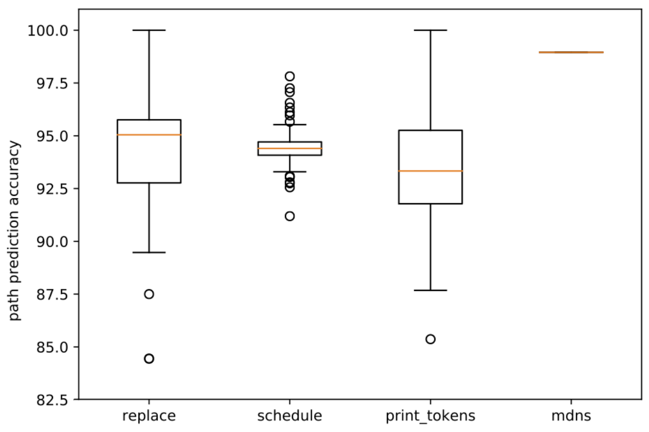

Fig. 3: Detailed path prediction accuracy for PG.

an outlier. In general, some limitations of the prediction technique that can affect the results (e.g., the one we mentioned earlier related to the use of a fixed sampling window). Furthermore, we found that different m2m paths may sometime result in instruction sequences that are very similar to each other, even if they execute different parts of the program, and thus generate EM emissions that are also very similar to each other [22]. For example, two different m2m paths may have the exact same mix of store and ALU operations. For another example, switch statements are commonly compiled into jump tables, which leads to multiple paths generating EM emissions that are difficult distinguish (a case that happens often for print_tokens and mDNS and causes a drop in accuracy).

D. RQ2: Comparison with ZOP

ZOP and ZOP-2 produce to some extent apples and oranges, as ZOP 's path profiles and ZOP-2 's complete traces are not directly comparable. To generate complete traces with ZOP we would have to extend the approach—and basically re-invent ZOP-2. We can however compare the path profile prediction accuracy of ZOP with that of ZOP-2 by extracting acyclic-path profiles from complete traces. Table III shows these results, computed for the traces generated using UC and PG and for the three benchmarks for which we have ZOP results. As the table shows, ZOP-2's accuracy is lower than but comparable to that of ZOP for replace and schedule (less than one percentage point), and slightly lower for print_tokens (around 6.5 percentage points).

The main reason for this slight decrease in accuracy lies in the fact that ZOP used a stateful model that selects the best match from m2m paths reachable from the currently predicted marker. This strategy reduces mispredictions, but is problematic when a misprediction does occur (as we discussed in Section III-C). Conversely, ZOP-2's stateless prediction suffers no additional penalty for mispredictions, which makes it effective for predicting complete paths, but can result in a larger number of individual sub-paths being mispredicted.

E. RQ3: Coverage

To answer RQ3, we measured the m2m path coverage achieved on all the benchmarks by the four input generation techniques considered. Table IV shows the results, together with the total number of m2m paths determined by static analysis. The PG and FN replay cases produced identical coverage, so their results are shown together.

TABLE III: Acyclic-path profiles prediction accuracy (%).

Benchmark	**ZOP**	**ZOP-2 (UC)**	**ZOP-2 (PG)**
replace	94.70	94.11	94.11
schedule	95.10	94.71	94.96
print_tokens	97.90	91.36	91.33

As the table shows, PG/PN achieved higher m2m path coverage than UC, which in turn achieved higher coverage than CS. This result is not surprising, as increasing symbolic state should necessarily lead to higher coverage. A more interesting question is how many additional *feasible* m2m paths were covered by PG/PN. Unfortunately, we cannot compute this information automatically, as it is an undecidable problem, and doing it by hand would be extremely time consuming and error prone. We can however compute a lower bound for this information by checking how many of the m2m paths covered by the evaluation inputs in our study (i.e., in the actual executions that we used to evaluate prediction accuracy) were missed by the generated input sets used for training; a decreasing number of uncovered m2m paths would necessarily indicate an increased number of feasible paths covered. Table V reports this information and clearly shows that PG/FN (i.e., PSE) consistently covers additional *feasible* subpaths that UC does not cover, and so does UC with respect to CS. This result provides initial indication that it is worth pursuing more aggressive state unconstraining approaches when generating inputs. However, more research and experiments are needed to demonstrate that client techniques can indeed benefit from the additional coverage achieved.

V. Threats to Validity

In this section, we briefly discuss the main threats to the validity of our empirical evaluation and steps we took to miti¡gate them. The main threat to internal validity is the potential for defects in our implementation. In mitigation, we based our implementation on KLEE, a reliable and stable symbolic execution engine. We also carefully tested our implementation of PSE, which is available for public inspection [4]. Threats to external validity include the size and number of our benchmarks. As discussed in Section IV-B, the maximum size of a potential benchmark was unfortunately constrained by the limitations of the embedded processor used for our evaluation. (Programs larger than mDNS could not be loaded onto the FPGA board.) Similarly, long signal recording, measurement, analysis, and human checks limited the number of benchmarks. Other threats to external validity are the way we selected inputs, especially for mDNS, and the possible lack of generalizability of our results to other devices. We are cognizant of these threats and plan to address them in future work by performing additional experiments on additional platforms.

TABLE IV: Coverage comparison among CS, UC, and PG/FN.

Benchmark	M2M Paths			
	total	CS	UC	PG/FN
replace	229	168 (73.4%)	179 (78.2%)	206 (90.0%)
schedule	153	119 (77.8%)	136 (88.9%)	149 (97.4%)
print_tokens	153	115 (75.2%)	132 (86.3%)	143 (93.5%)
mDNS	5763	509 (8.8%)	2454 (43.3%)	4147 (72.0)%

TABLE V: Missing m2m paths in the training inputs.

Benchmark	CS	UC	PG/FN
replace	3	2	0
schedule	7	2	0
print_tokens	9	3	0
mDNS	96	18	0

VI. Related Work

Prior approaches to program tracing required instrumentation of the monitored system or runtime support. Software instrumentation is unfortunately expensive (e..g, they can incur 31% overhead for (acyclic) path profiling alone [2]). Recent processor designs, such as Intel Processor Trace, can reduce this overhead to as little as 5% [23], but they require a sophisticated processor, still entail non-zero overhead, and consume considerable storage capacity and throughput.

The key concepts of classical symbolic execution, as provided in [24], have been implemented for input generation in many prior tools [9], [18], [25]–[32]. Our input generation work builds on this rich body of research on symbolic execution and automated test generation.

Our PSE technique is closely related to UC-KLEE [33], which performs under-constrained symbolic execution from an arbitrary function call. This degree of under-constraining has been shown to be effective for patch validation and defect detection [3]. However, unlike PSE, UC-KLEE misses some subpaths needed for complete ZOP-2 training.

Chopped symbolic execution [34] shares with our work the goal of reaching code buried deep in the call graph, but takes an orthogonal approach; it employs program slicing to exclude uninteresting portions of the code from symbolic execution, while maintaining soundness. For ZOP-2 training, the only uninteresting code is dead code, so this approach would not be applicable in our context.

Since symbolic execution is susceptible to path explosion, numerous approaches have been defined to address this issue. Guided path search heuristics select paths for exploration either randomly [18], [35] or based on the predicted likelihood of reaching a given coverage target [18], [25], [35]. Other approaches reduce the number of paths to search by removing equivalent paths [36], removing paths that cannot reach new code [37], or merging state on selected paths [38]. PSE could benefit from (suitably adapted versions of) these techniques.

Lazy initialization is an important feature of generalized symbolic execution [9], as it allows the handling of unconstrained pointers or references. Various tools have implemented lazy initialization for symbolic execution in Java (e.g., [9], [39], [40]), C/C++ (via LLVM IR [3]), and object code [41]. Because this technique can inflict a significant performance penalty, researchers have proposed optimizations for Java-based symbolic execution [42]–[44]. However, these optimizations would be difficult to implement without Java's memory manager and strict type safety. UC-KLEE [33] uses a version of lazy initialization that models unconstrained pointers as either NULL or pointing to a new memory object. We augment this model with existing memory locations that were either allocated with or typecast to the type of entity being lazily initialized.

Unintentional EM information leakage has been traditionally exploited by attackers for cryptographic key extraction from target systems. More recently, researchers have leveraged EM side channels for hardware Trojan detection [45], [46], malware detection [47], control flow integrity assessment [48], and software profiling [1], [49]. Sehatbakhsh and colleagues exploit spectral peaks in EM signals for loop-level profiling of program executions, while the approach by Callan and colleagues [1] can predict acyclic path profiles. Unlike this previous work, our approach can not only profile program executions at the basic-block level, but also predict the entire control flow path with high accuracy.

VII. Conclusion and Future Work

We presented ZOP-2, a new approach that can collect complete execution traces accurately and with zero overhead by leveraging EM emanations. ZOP-2 can greatly benefit and support several important developer tasks, such as debugging applications, tuning performance, and profiling users. Our evaluation results show that ZOP-2 is indeed effective and can produce accurate execution traces without requiring any program instrumentation.

In future work, we will enhance our signal processing technique so as to improve its performance when operating on small code fragments and when m2m paths and sampling windows are not aligned. We will also investigate the use of additional signal processing techniques that will allow us to handle more sophisticated processor architectures and perform evaluations on larger benchmarks. Although stateless signal matching allows ZOP-2 to recover from mispredictions, it also causes some additional mispredictions. We believe that a mediated use of control flow information may allow for increasing accuracy while still permitting recovery after mispredictions. Finally, we will study applications of ZOP-2 and PSE in different contexts, including (1) for malware detection, when the observed hardware executes unsanctioned code, and (2) for fault detection, where the additional coverage provided by state unconstraining may reveal faulty behaviors missed by other input-generation techniques.

Acknowledgments

This work was partially supported by NSF, under grants CCF-1161821 and 1563991, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, ONR, under contract N00014-17-1-2895, and gifts from Google, IBM Research, and Microsoft Research.

References

[1] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso, "Zero-overhead Profiling via EM Emanations," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 401–412.

[2] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, 1996. MICRO-29*, Dec. 1996, pp. 46–57.

[3] D. A. Ramos and D. Engler, "Under-Constrained Symbolic Execution: Correctness Checking for Real Code," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64.

[4] "Zop-2," https://hub.docker.com/r/naegling/icse19-zop2.

[5] R. Rutledge, S. Park, H. Khan, A. Orso, M. Prvulovic, and A. Zajic, "Artifact: Zero-Overhead Path Prediction with Progressive Symbolic Execution," Feb. 2019, doi: 10.5281/zenodo.2577372.

[6] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[7] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[8] D. Engler and D. Dunbar, "Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 1–4.

[9] S. Khurshid, C. S. PĂsĂreanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, Apr. 2003, pp. 553–568.

[10] B. B. Yilmaz, R. L. Callan, M. Prvulovic, and A. Zajić, "Capacity of the em covert/side-channel created by the execution of instructions in a processor," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 605–620, March 2018.

[11] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Apr. 2002, pp. 213–228.

[12] "Clang C Language Family Frontend for LLVM," https://clang.llvm.org/.

[13] "NetworkX — NetworkX," https://networkx.github.io/.

[14] "KLEE," http://klee.github.io/.

[15] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[16] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds. Springer Berlin Heidelberg, 2007, pp. 519–531.

[17] "Matlab," http://www.mathworks.com/products/matlab.html.

[18] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[19] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[20] "Avahi - service discovery for linux using mdns/dns-sd," http://www.avahi.org/.

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

[22] R. Callan, A. Zajić, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 242–254. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.39

[23] A. Kleen and B. Strong, "Intel processor trace on linux," *Tracing Summit*, vol. 2015, 2015.

[24] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[25] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1–10:38, Dec. 2008.

[26] C. Cadar and D. Engler, "Execution Generated Test Cases: How to Make Systems Code Crash Itself," in *Model Checking Software*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Aug. 2005, pp. 2–23.

[27] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[28] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012.

[29] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272.

[30] N. Tillmann and J. de Halleux, "Pex–White Box Test Generation for .NET," in *Tests and Proofs*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Apr. 2008, pp. 134–153.

[31] H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara, "FSX: Fine-grained Incremental Unit Test Generation for C/C++ Programs," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 106–117.

[32] G. Li, I. Ghosh, and S. P. Rajan, "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jul. 2011, pp. 609–615.

[33] D. A. Ramos and D. R. Engler, "Practical, Low-Effort Equivalence Verification of Real Code," in *Computer Aided Verification*. Springer Berlin Heidelberg, Jul. 2011, pp. 669–685.

[34] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped Symbolic Execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 350–360.

[35] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 443–446.

[36] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Mar. 2008, pp. 351–366.

[37] S. Bugrara and D. Engler, "Redundant State Detection for Dynamic Symbolic Execution," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 199–212.

[38] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient State Merging in Symbolic Execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 193–204.

[39] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF–SE: A Symbolic Execution Extension to Java PathFinder," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Mar. 2007, pp. 134–138.

[40] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test Input Generation with Java PathFinder," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 97–107.

[41] P. Godefroid, "Micro Execution," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 539–549.

[42] N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser, and M. F. Frias, "BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 639–660, Jul. 2015.

[43] P. Braione, G. Denaro, and M. Pezzè, "Enhancing Symbolic Execution with Built-in Term Rewriting and Constrained Lazy Initialization," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 411–421.

[44] X. Deng, J. Lee, and Robby, “Efficient and formal generalized symbolic execution,” *Automated Software Engineering*, vol. 19, no. 3, pp. 233–301, Sep. 2012.

[45] J. Balasch, B. Gierlichs, and I. Verbauwhede, “Electromagnetic circuit fingerprints for hardware trojan detection,” in *2015 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, Aug 2015, pp. 246–251.

[46] O. Söll, T. Korak, M. Muehlberghuber, and M. Hutter, “Em-based detection of hardware trojans on fpgas,” in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, May 2014, pp. 84–87.

[47] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, “Eddie: Em-based detection of deviations in program execution,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: ACM, 2017, pp. 333–346. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080223

[48] Y. Han, S. Etigowni, H. Liu, S. Zonouz, and A. Petropulu, “Watch me, but don’t touch me! contactless control flow monitoring via electromagnetic emanations,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. New York, NY, USA: ACM, 2017, pp. 1095–1108. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134081

[49] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic, “Spectral profiling: Observer-effect-free profiling by monitoring em emanations,” in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–11.