Zero-Overhead Profiling via EM Emanations

Robert Callan[1], Farnaz Behrang[2], Alenka Zajic[1], Milos Prvulovic[2], Alessandro Orso[2]

[1]School of Electrical and Computer Engineering, Georgia Tech, USA
rcallan@gatech.edu, alenka.zajic@ece.gatech.edu

[2]School of Computer Science, Georgia Tech, USA
behrang@gatech.edu, milos@cc.gatech.edu, orso@cc.gatech.edu

ABSTRACT

This paper presents an approach for zero-overhead profiling (ZOP). ZOP accomplishes accurate program profiling with no modification to the program or system during profiling and no dedicated hardware features. To do so, ZOP records the electromagnetic (EM) emanations generated by computing systems during program execution and analyzes the recorded emanations to track a program's execution path and generate profiling information. Our approach consists of two main phases. In the training phase, ZOP instruments the program and runs it against a set of inputs to collect path timing information while simultaneously collecting waveforms for the EM emanations generated by the program. In the profiling phase, ZOP runs the original (*i.e.,* uninstrumented and unmodified) program against inputs whose executions need to be profiled, records the waveforms produced by the program, and matches these waveforms with those collected during training to predict which parts of the code were exercised by the inputs and how often. We evaluated an implementation of ZOP on several benchmarks and our results show that ZOP can predict path profiling information for these benchmarks with greater than 94% accuracy on average.

CCS Concepts

•Software and its engineering → Software performance;

Keywords

Profiling, performance, PLEASE ADD MORE

1. INTRODUCTION

Program profiling is a type of dynamic analysis that measures some aspects of software behavior. One of the most common instances of program profiling counts the execution of instructions or sequences of instructions and uses that information to identify heavily executed paths (also called *hot paths*). Knowledge of the hot paths can guide other tasks such as code optimization and performance analysis. Profiling is typically implemented by adding software probes (instrumentation) to a program's source code or binary executable and these probes either log events of interest or update statistics about such events at runtime.

This approach is effective in many usage scenarios, but there are a few exceptions. Adding instrumentation unavoidably adds runtime and resource overheads. Runtime overheads can alter the timing of events, and so in real-time systems or cyber-physical systems these timing changes can affect the path taken through the profiled program. In fact, if overheads are high enough, these systems may fail (*e.g.,* miss real-time deadlines) if they are profiled under "in the field" conditions. Profiling is also challenging in already-deployed software [32], where deployed systems (that for example suffer unexplained performance problems) would ideally be profiled *in situ* to ensure that the profiling results capture the actual program behavior in that deployment. Hardware features can minimize (though rarely eliminate) the software overhead of detailed profiling. They are also costly in terms of chip/PCB space and development time and feature support varies between devices. Profiling embedded controllers presents additional challenges since these devices often lack sufficient memory space to store the extra code (instrumentation) and profiling-related data structures, and sometimes lack the I/O interfaces to report the profiling results back to the programmer.

An ideal profiling solution would be one that gathers (1) perfectly accurate information about what is actually executed during profiling (2) without changing anything about the profiled system: no code instrumentation, no data structures for profiling information, no additional I/O activity, and no changes to the hardware of the system. While instrumentation can provide perfectly accurate profiling information, it is an inherently intrusive technique that – even when minimal and designed so as not to affect the semantics of the instrumented code – changes some important aspects of the code's dynamic behavior.

In contrast, this paper proposes a technique, which we call ZOP (Zero-Overhead Profiling). ZOP retains the second aspect of ideal profiling (no changes to the profiled code or system) at the cost of less-than-perfect accuracy. ZOP computes profiling information in a highly accurate and completely non-intrusive way by leveraging electromagnetic (EM) emanations generated by a system as the system executes code. Because ZOP generates profiling information without interacting with or modifying the profiled system, it offers the potential to profile a variety of software systems for which profiling was previously not possible. The ability to collect profiles by placing a profiling device next to the system to be profiled provides some appealing advantages over traditional instrumentation-based approaches in many traditional contexts as well.

ZOP first measures the EM emanations produced by the system to be profiled as the system processes inputs whose execution path is known (the *training* phase). This allows ZOP to build a mo-

del of the waveforms produced by different code fragments. ZOP then collects emanations from a new, to-be-profiled (*i.e.,* unknown) execution and infers which parts of the code are being executed at which times (the *profiling* phase). This inference is accomplished by matching the observed unknown emanations to emanations from the training phase which are known to be generated by particular code fragments.

The main contributions of this paper are:

- ZOP, a completely non-invasive profiling approach, where profile information is inferred from EM emanations of the (unmodified) system as it runs the (unmodified) to-be-profiled software.
- A proof-of-concept implementation of ZOP that shows that our approach is practically feasible.
- Empirical results that (1) show that ZOP can achieve high profiling accuracy and (2) provide insight into the performance of ZOP that suggest directions for further research.

The rest of the paper is organized as follows. Section 2 describes at a high level how program execution can be related to EM emanations. Section 3 describes how ZOP generates a training model and uses EM emanations along with this training model to generate profiling data for new program executions. Section 4 describes an implementation and experimental evaluation of ZOP. Section 5 describes threats to this work's validity, Section 6 describes related work, and Section 7 presents conclusions and future work.

2. BACKGROUND

Computing devices generate electromagnetic (EM) emanations when they operate. While previous research has demonstrated that useful information about a system's behavior may be embedded in these emanations [3, 13, 24], it also suggests that such information extraction on devices with highly optimized microarchitecture is difficult in practice. Nearly all existing techniques for extracting information from EM emanations are used for side channel analysis in cryptography and so are focused on extracting information about *a value used by the program*, such as a cryptographic key. Furthermore, these techniques operate in an adversarial context, *i.e.,* they must overcome program and hardware features (countermeasures) that are specifically designed to mask or obfuscate the impact that the desired data values have on EM emanations.

Profilers have a few advantages that side-channel attackers do not have. The profiled system is cooperative, so there are no countermeasures and the profiler may position probes wherever needed to get the best EM signal. Also, program profilers record statistics about when and how often parts of a program execute and are not primarily focused on data values. Sequences of instructions and control flow decisions affect EM emanations more strongly than changes in data values do, potentially making profiling information easier to extract than data values.

While the details of how computing devices generate EM emanations are outside the scope of this paper, a brief example describing the EM emanations produced by a processor's clock may provide some helpful insight into the connection between EM emanations and program behavior. Each cycle of a processor's clock the processor's state is updated, generating a current at the processor clock frequency. Conceptually, the amplitude of this current depends on how much of the processor's state changes each clock cycle and therefore the current depends strongly on which instructions are active or recently executed. As a program executes, the processor executes different instructions based on control flow decisions, and this variation in instruction execution therefore modulates the amplitude of the processor clock current. EM emanations from the processor can be directly related to the current drawn by the processor. These phenomena together create a direct link between the processor clock EM emanations and program behavior. ZOP uses this link to determine which code executes and how frequently.

If a program executes several times with the same inputs, the processor clock EM emanations waveforms recorded during program execution may vary significantly between program runs, but in general these variations are small compared to the waveform features observed in every execution. EM noise from other devices, radio broadcasts, or communications signals can cause these run-to-run variations. However by demodulating the signal at the frequency of the processor clock, we are filtering out any noise outside of the narrow band of the RF spectrum around that clock frequency. Furthermore, specially designed EM probes and signal processing can be used to filter out noise with properties distinguishable from our signal of interest (*e.g.,* eliminate noise and signals not generated by the processor). In addition to external noise, system activity unrelated to the program and the accumulation of small timing differences caused by the complexity of the system (*e.g.,* cache and memory behavior) create run-to-run variations between repeated executions with the same inputs. However, these variations are usually smaller than the waveform differences created by execution of different paths through the program and so by observing a sufficient number of dynamic instances of the same static path, we can later recognize this path by matching it against one of its dynamic instances. For example, if a short path has two dynamic instances, one with a cache miss and one with a cache hit, we can recognize this path has long as we have examples of both possible dynamic instances. We will explain in Section 3.3 how the ability to recognize short paths can be used to predict the path through whole programs.

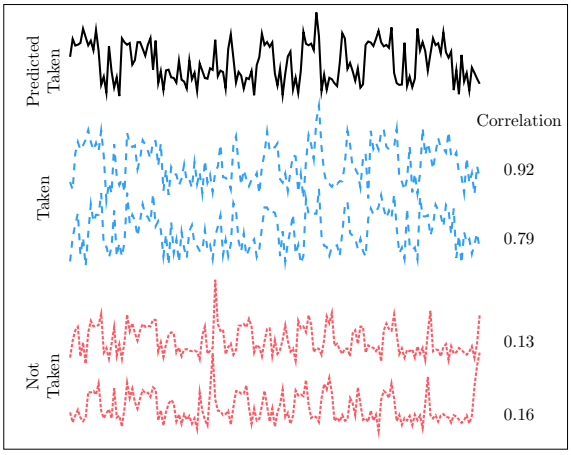

Figure 1: Examples of waveforms collected by measuring EM emanations produced by several executions.

Figure 1 shows several waveforms recorded during a short fragment of program execution. All of these waveforms start at the same static location in the program, and each follows one of two paths depending on whether the $true$ or the $false$ path of a conditional statement is followed. In particular, the dashed waveforms correspond to execution along the $true$ (conditional branch instruction is "taken") path, whereas the two dotted waveforms correspond to execution along the $false$ path (branch instruction is "not taken"). Assume we use dynamic analysis to determine whether the branch is taken for these cases. It is clear from Figure 1 that while there are some differences between these "training" waveforms that correspond to the same path, these differences are smaller than those between the $true$ and $false$ paths. To determine

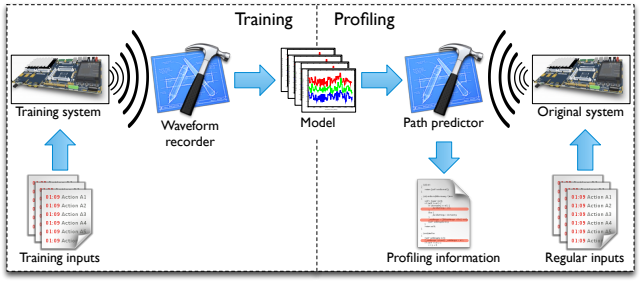

Figure 2: High-level view of our approach.

which path was taken in the "unknown" (solid) waveform without doing any dynamic analysis, we calculate the correlation coefficient between that unknown waveform and each of the candidate training waveforms. By observing correlation coefficients, we are able to determine with high confidence that the branch was taken in the unknown execution since the branch-taken training examples correlate much better than the branch-not-taken examples.

In the next section, we (1) introduce the ZOP approach, (2) describe how we can create a model that encodes the features of the waveforms we collect during training and (3) use this model to predict the path taken during an unknown execution using only the waveform produced by this execution without using *any* runtime instrumentation.

3. THE ZOP APPROACH

The goal of ZOP is to compute code profiling information without any instrumentation. Figure 2 shows a high-level overview of our approach. As the figure shows, ZOP has two main phases. In the *training phase*, ZOP runs instrumented and uninstrumented versions of the program against a set of training inputs, records EM emanations for these executions, and builds a model that associates the recorded waveforms with the code subpaths that generated them. In the *profiling phase*, ZOP records the EM waveform generated by an execution of a vanilla (*i.e.,* completely uninstrumented) version of the program, finds the closest match between these waveforms and the waveforms in the training model, and predicts that the unknown execution takes the same path as the subpaths taken during the matching training waveforms. ZOP implements these two high-level phases in the steps and substeps in the fairly complex workflow shown in Figure 3. In the next sections, we explain the different steps and substeps of this work flow in detail.

3.1 Training 1

The left part of Figure 3 shows the Training 1 phase of ZOP approach. During Training 1, we run an instrumented version of the system against a set of training inputs. This step is needed to reconstruct a graph model of the program's states, to determine the timing of each subpath, and to establish the correspondence between subpaths and the EM waveforms they generate. We refer to the instrumentation points as "markers" since they are used to "mark" the time of each executed instrumentation point in the EM waveform. In order to ensure optimal placement of these markers for generating accurate profiling information, the level of granularity of the inserted instrumentation points (markers) is critical.

In general, matching the EM emanations waveform from an unknown execution path to example waveforms for known execution paths is not a simple task. Matching complete program executions is clearly not an option, as it would require observing all possible executions to build a model. An ideal model would, in fact, be one that learns the waveform for each processor instruction independently since this would make recognition easiest. Some recent research matches waveforms on an instruction by instruction basis [38, 46] for non-profiling applications but this technique has only been applied to the simplest of processors and has not yet been successfully applied to path profiling.

We contend that longer subpaths must be considered for this matching to be successful in more complex processors where superscalar, out-of-order microarchitecture and variable latency memory interfaces make instruction by instruction recognition impractical. Therefore, in our approach, we consider acyclic paths, as defined by Ball and Larus [7], as the basic profiling unit. (Intuitively, acyclic paths are subpaths within a procedure such that every complete path in the procedure can be expressed as a sequence of acyclic paths.) In other words, ZOP learns the waveforms generated by the execution of acyclic paths exercised by the training inputs and then tries to recognize these paths based on their waveforms during profiling. The acyclic paths provide a level of the granularity that simultaneously (1) keeps the marker to marker paths short enough that a reasonable number of training examples can represent all the possible marker to marker waveform behaviors and (2) keeps the training instrumentation overhead low enough that the instrumentation itself does not drastically affect the execution waveforms.

The ***Instrumenter*** module starts by computing the acyclic paths in the code as defined by Ball and Larus [7]. For every identified path in the source code, it selects a subset of instrumentation locations that is used in the Ball and Larus approach to uniquely identify the path (mainly the beginning and end of the path) and adds markers to those locations in the user's source code. Thus the instrumentation locations are similar to approaches to lightweight program tracing such as [40].

The example code shown in Figure 4 consists of a C function called `putsub` that is a slightly simplified version of a function present in one of the programs we used in our evaluation (see Section 4). Marker positions for this example function are shown in Figure 5. Each time a `marker()` instrumentation point is encountered, the marker (*e.g.,* A,B,C, etc.) and the time elapsed since the start of the program are recorded to an array. To illustrate the instrumentation with an example, one execution of the `putsub()` function may take the path ABDEF. The recorded values then show the time when A was encountered, followed by the time when B was encountered, etc. For each training input we run the instrumented code and record the EM waveform. We can then "mark" the EM waveform with the current program location at each time a marker is encountered. For example we could find all the start and end times for the instances of the AB subpath in the training executions and extract the portions of the EM waveforms for these times. It is important to re-emphasize that instrumentation is only used during the Training 1 phase. The program profiled during the Profiling phase is unmodified and uninstrumented.

```
1  void putsub(char* lin, int s1, int s2,
       char* sub) {
2    int i = 0;
3    while (sub[i] != ENDSTR) {
4     if (sub[i] == DITTO) {
5       int j = s1;
6       while (j < s2)
7         fputc(lin[j++], stdout);
8     } else
9       fputc(sub[i], stdout);
10    i++;
11   }
12 }
```

Figure 4: Uninstrumented `putsub()` function.

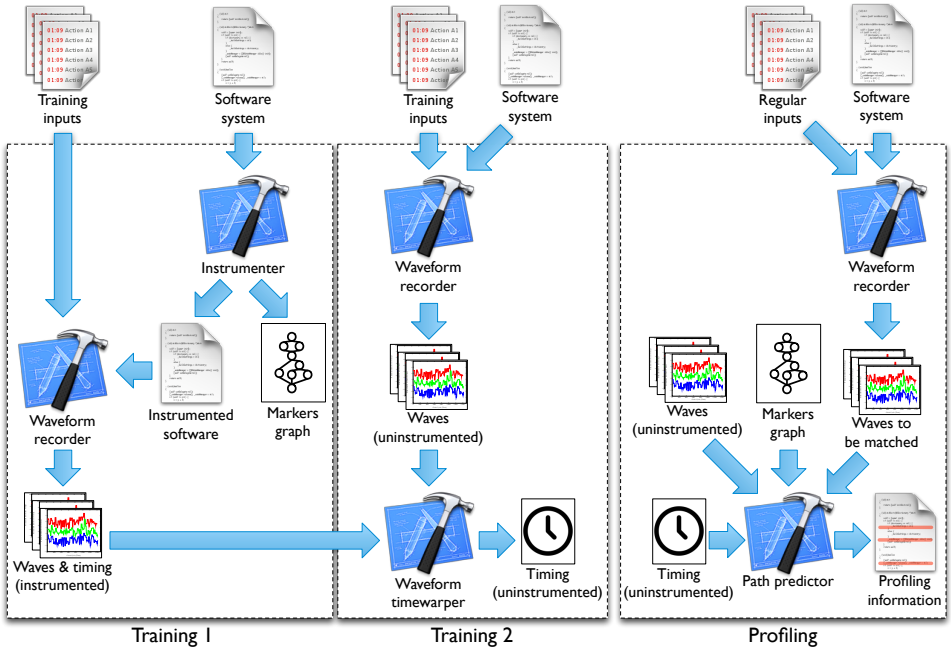

Figure 3: Workflow of ZOP. (Note that we repeat some elements to reduce clutter, improve clarity, and better separate the different steps of the approach; that is, multiple elements with the same name represent the same entity.)

```
1  void putsub(char* lin, int s1, int s2,
       char* sub) {
2    int i = 0;
3    marker(A);
4    while (sub[i] != ENDSTR) {
5      marker(B);
6      if (sub[i] == DITTO) {
7        int j = s1;
8        while (j < s2) {
9          marker(C);
10         fputc(lin[j++], stdout);
11       }
12     } else {
13       marker(D);
14       fputc(sub[i], stdout);
15     }
16     i++;
17     marker(E);
18   }
19   marker(F);
20 }
```

Figure 5: Instrumented `putsub()` function.

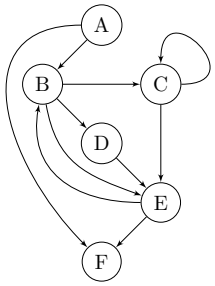

Figure 6: The marker graph for the `putsub()` example.

The ***Markers Graph*** models the possible paths between marker code locations. As an example, Figure 6 shows a graph derived from the `putsub()` function in Figure 5. The graph's nodes are the markers for `putsub()`, and a directed edge occurs from marker X to marker Y if the program can reach Y from X without reaching another marker in between. While this graph shows a single edge between X and Y, there may be thousands of training examples for each such two marker subpath. Therefore, to predict the whole execution path, we need to not only predict the next marker but also the time the execution took to get from X to Y.

The ***Waveforms and Timing*** block of Training 1 contains the recorded waveform examples for subpaths in the program where the correspondence between an execution's waveform and the code path taken is known, but these waveforms are affected by the computations done by the instrumentation so they are not suitable for matching to uninstrumented code during profiling. Thus our next step is to collect waveforms for the same training inputs, this time without instrumentation, and identify the times in these instrumentation-free waveforms that correspond to marker positions in the code (even though the uninstrumented code has no markers at these positions).

3.2 Training 2

The middle of Figure 3 shows the Training 2 phase of the ZOP approach. In this phase we run an uninstrumented version of the code with the same set of inputs used in Training 1, collect the waveforms for these executions, and perform matching to determine the points in these new waveforms that correspond to marker positions in the corresponding waveforms from Training 1. This results in waveforms generated by uninstrumented execution, but in which we do not know which part of the waveform corresponds to which marker-to-marker part of the program code. These waveforms must be compared to those observed during profiling to infer which part of the code is executing at each point in the profiling run. To do this, we must infer the timing of the uninstrumented code, *i.e.,* we must determine which part of the instrumentation-free (Training 2) signal corresponds to which part of the instrumentation-marked (Training 1) signal and thus, transitively, to determine which portions

of the waveforms collected during profiling correspond to which subpaths in the instrumentation-free program code.

This two-phase training approach has the key property that while the device/environment used for Training 2 must be similar to that used for Profiling, the device/environment used for Training 1 can differ from that used for Training 2 and Profiling. For example, while Training 1 requires the same compiler, optimization, and general processor architecture as the other phases but it does not require the exact same processor or system design. Therefore a user could, for example, perform Training 1 on a development board with more resources and flexibility to facilitate the required instrumentation, and then Training 2 and Profiling could be done on a production system which does not have the resources or flexibility to handle instrumentation since neither of these phases requires instrumentation. Then Training 2 could be done on a production system by software developers and Profiling could be done on by system users in the field. Furthermore, future work may allow ZOP to skip Training 1 altogether by deriving the timing information (marker times) via signal processing and machine learning.

3.2.1 *Inferring Timing for the Uninstrumented Code Using Time Warping*

The key to identifying which uninstrumented (Training 2) waveform corresponds to which part of the code is that, for each training input, we have executed the code twice, once with the instrumented program and once with the uninstrumented program. This means that the path through the code is the same for both executions, and that the EM signals for the two executions will tend to be similar at points that correspond to execution between markers, but one of the signals (the one from Training 1) has additional (marker instrumentation) activity inserted, along with some distortion of the signal at the transitions between instrumentation and "real" program activity. An example matching between instrumented and uninstrumented execution waveforms for the same training inputs is shown in Figure 7. The longer red waveform (at the top of the figure) corresponds to execution of the instrumented code, and the vertical solid black lines show the (known) timing of the markers as recorded by instrumentation. The shorter waveform (at the bottom of the figure) corresponds to uninstrumented execution, where the timing of the markers is not known because this execution has no instrumentation. Note that the instrumented and uninstrumented waveforms share many of the same features, but that there are also significant differences (*e.g.,* the DE and BC paths for example). These differences are often larger than the differences between two unique dynamic instances of the same subpath, so profiling accuracy would be poor if we simply use (instrumented) waveforms from Training 1 to match to signals collected during (uninstrumented) profiling.

To systematically determine which part of the Training 1 signal corresponds to which part of the Training 2 signal for the same input, a technique such as dynamic time warping [43] can be used. More generally time warping between two signals can cut out parts of the top signal (shifting later samples of this signal to fill the gap made by the cut-out) in such a way that the remaining samples of the top signal are as similar as possible to the bottom signal. After time warping we know which points in the instrumentation-free waveform corresponds to the marker points in the instrumented-run waveform, as shown by the dotted lines in Figure 7.

3.3 Profiling

The right column of Figure 3 shows the Profiling phase of ZOP. In the Profiling phase, we run the uninstrumented program with the to-be-profiled inputs, record the EM waveforms produced, and compare these waveforms to the waveforms collected (and annotated with marker information) in Training 2.

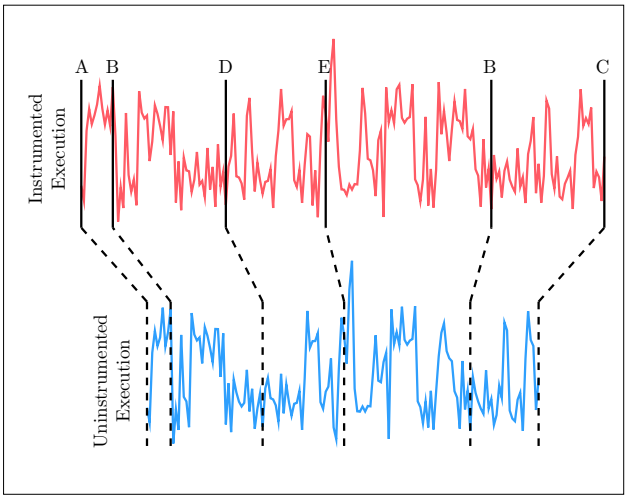

Figure 7: Estimating path timing in uninstrumented training executions using waveform time warping.

3.3.1 *Path Predictor*

The Training 1 and 2 phases of ZOP yield waveforms and marker timing information for the set of training inputs used in the uninstrumented program as well as the markers graph. When a particular short subpath occurs during the profiling program execution, the resulting waveform will be similar to a training waveform of that same short subpath. To predict, for example, the execution path taken by the `putsub()` function, we run the uninstrumented version of `putsub()` with a to-be-profiled input and record the waveform shown at the bottom of Figure 8.

To illustrate how our *Path Predictor* works, here is one example. For the profiling waveform shown in Figure 8, we start with no information about the path taken. According to the markers graph, the profiling execution must start with marker A at the beginning of the waveform. The next marker encountered can be either B or F according to the marker graph. We use the Pearson correlation coefficient [50] to compare the profiling waveform with the three training waveforms in Figure 8. All three training waveforms start with an AB subpath which very closely matches the start of the profiling waveform. Although it is not shown, assume that we have another training example with the AF path and this AF waveform does not match the profiling waveform. Then we can infer that the profiling execution starts with the AB path and that B occurs at the same time in the execution as it does in the training executions. There are two possible next subpaths from B, either BD or BC. Examining all the training waveform sections for BD and BC, it is clear that the profiling waveform matches the BD section in the top training waveform more closely than it does the BC section in the third training waveform. Therefore we can infer that the profiling execution takes the BD path. From D the only possible next marker is E, so we find the most closely matching DE waveform and update our predicted path to ABDE. From E the code encounters either F or B next. Comparing the EF and EB waveforms, it is clear the profiling execution has taken the EB path next. We repeat this waveform matching and path updating process until we reach the exit marker F. This process predicts the ABDEBCEF path.

Figure 8 and its description captures the essence of the training and path prediction algorithm but some refinements are needed to

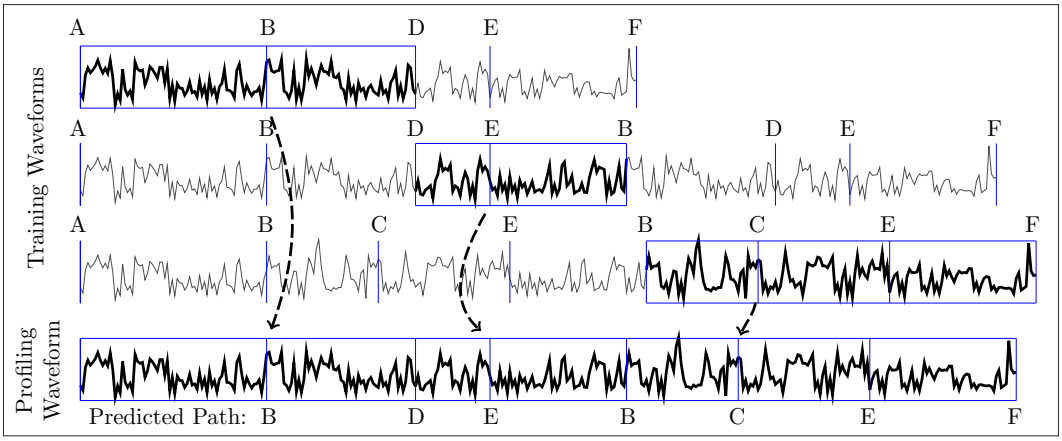

Figure 8: Predicting the execution path of the `putsub()` example by matching training waveform segments to an execution waveform.

achieve adequate performance. Consider what happens when an incorrect prediction is made. For example, assume we incorrectly selected ABCE at the start of the profiling waveform instead of the correct path ABDE. In such a case not only is the subpath through C wrongly predicted but in addition even though we have predicted D correctly as the next marker, the time of the D marker is too early. When we match the training subpaths starting at D assuming this incorrect time for D, the training waveforms may no longer match the profiling waveform well. Blindly selecting the most closely matching next subpath is not guaranteed to result in the most closely matching waveforms for the entire execution. Such errors tend to compound and the predicted execution path may diverge from the actual execution path indefinitely. This issue may be even worse when an incorrect marker is predicted and the predicted path and the actual path diverge for a long time following the incorrect decision. To address these issues we need to model the search for the optimal execution path more precisely.

3.3.2 *Path Prediction as a Tree Search*

When we reach a marker X at a particular time t in the profiling waveform we compare all the training subpath waveforms starting at X against the profiling waveform starting at t and assign a score to each training example. We use the correlation coefficient as the similarity metric between the section of the profiling waveform starting at t and the training subpath waveform. Therefore for each training example we get a correlation value, a next marker, and the time of the next marker (*i.e.,* the start time t plus the duration of the training subpath).

We can think of the search for the optimal execution path through the program as a tree search. The root node is the entry marker (marker A in Figure 5) and each child node has an edge for each training subpath example starting at that node's marker. Each node in this tree has a marker (*e.g.,* A, B, C, etc.) and a starting time t in the profiling waveform. Each edge corresponds to a single training subpath example waveform and has three properties: a duration (the duration of the training example), a correlation between the training subpath waveform and the profiling waveform starting at time t, and the marker at the end of the subpath in this training example. According to these definitions a search tree for an example execution waveform of `putsub()` can be made as shown in Figure 9. Each edge in the figure denotes a training example subpath and its waveform. The edge weights shown are the correlation values for each edge's training example (only the highest correlated subpath edges are shown).

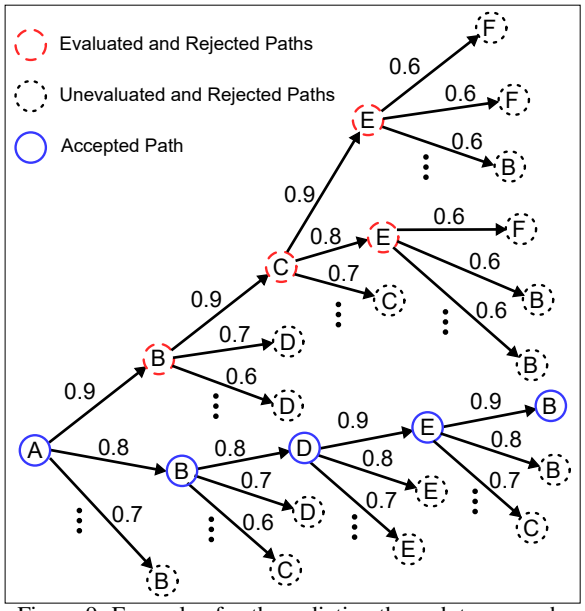

Figure 9: Example of path prediction through tree search.

The branching factor for these trees is large because each node may have thousands of training examples. To simplify the search, we employ the following heuristic. The heuristic's goal is to find a root-to-leaf path whose edges all have correlation greater than a chosen value C_{th}. To evaluate a node, we calculate the correlation of each next edge and sort the nodes in order of decreasing correlation. If the edge with the maximum correlation is greater than C_{th} we continue searching along this edge. Otherwise, we indicate this node as rejected and backtrack along our path so far (*i.e.,* toward the root node). As we backtrack we stop at the first node that has an edge to an unevaluated node with correlation greater than C_{th} and search forward along this edge. In Figure 9, $C_{th} = 0.75$ and the search algorithm follows the red dashed nodes from A to B to C to E along the top-most edges. No edge from this E exceeds C_{th} so the algorithm backtracks to C and then continues forward to the E with $C_{th} = 0.8$. No edge from here exceeds C_{th} so the algorithm backtracks along C to B to A and then moves forward along the accepted blue path. Because ZOP identifies paths by matching short subpaths between marker pairs and backtracks when unsuccessful, ZOP can recognize paths that it never observed during training.

This heuristic clearly results in a path where each edge is greater than C_{th} if such a path exists since we only follow edges with cor-

relation greater than C_{th}. It is not guaranteed that all paths that meet our selection criteria correspond to the correct whole program execution path however. Furthermore, it is not guaranteed that a path meeting the selection criteria exists. Many sophisticated heuristic algorithms exist for searches through trees with similar properties [10, 12, 31, 42], and it is expected that future research into this problem will greatly improve the overall path prediction accuracy.

Two minor refinements are required to make this algorithm practical. First, when correlating the training examples against the profiling waveform it is necessary to correlate the waveforms several times with slight misalignments between the training waveforms and the profiling waveforms and use the best result of all the alignments. This is because the current position in the program is always an estimate, so by trying several different alignments and selecting the best alignment we can keep track of the current program location with better accuracy. The second refinement is that the training waveforms for each edge are extended beyond the time position of the next marker so that all the training waveforms starting at a given marker have the same length. This is done by finding the training example for each marker with the longest duration and extending the other training example waveforms for this marker to the same length. This is required to allow fair comparisons between training examples which would otherwise have different lengths (shorter signals are more likely to be more highly correlated due to random chance than longer signals). This approach has the added benefit that (with some preprocessing) all the training waveforms for a given starting marker can be correlated (with several different alignments) against a profiling waveform using a single matrix multiplication which greatly reduces runtime.

3.3.3 Eliminating Search Paths

Removing nodes before they are evaluated can greatly decrease runtime because the evaluation of each node in this tree is expensive and the tree branches quickly. Some nodes can be rejected quickly without sacrificing much accuracy. For example, suppose two edges W and Z start at a node B and represent BD training waveforms with nearly identical durations. This repetition is common because executions of the same subpath often have roughly the same runtime. Suppose W has higher correlation to the profiling waveform than Z. We can immediately reject Z without evaluating it because the D node following W and the D node following Z occur at the same time in the profiling waveform (since W and Z start at the same time and have the same duration). If W is evaluated and rejected, evaluating Z would just re-evaluate an identical D node with nearly the same start time.

We can eliminate more edges by observing that many marker sequences do not correspond to valid execution paths. To see this, recall that the path prediction execution paths are interprocedural and that we allow an edge from any marker X to a marker Y if an X to Y transition is possible in the profiling program. Then consider a function which contains a single marker F. This function is called from two points A and B in the program, returning at points C and D respectively. Then the only valid marker sequences for this function call would be AFC and BFD. The algorithm described so far would however also evaluate the impossible paths AFD and BFC. Ideally, a fully constrained grammar of all possible paths would be generated to limit the search to possible marker sequences. This grammar could enumerate the set of valid next markers from any node in the search tree. This grammar would be difficult to generate, so instead we keep a function call stack for the currently evaluated execution path and any next marker which would be inconsistent with the call stack is rejected. Note this is a very weak constraint and only eliminates the impossible AFC and BFD sequences when A and B are contained in different functions.

Table 1: Benchmark statistics.

Benchmark	LOC	Markers	Training Set Size	Profiling Set Size
print_tokens	571	48	240	400
schedule	415	36	284	400
replace	563	54	299	400
Total	1549	138	823	1200

3.4 Profiling Information

In the final step of ZOP, we construct the paths for the profiling inputs from a set of predicted markers provided by the previous steps. Every consecutive pair of predicted markers represents a set of basic blocks that are executed between two markers by a training input. First, for every training input and every two consecutive pair of markers we extract the basic blocks that are executed between them. Once we collect the basic blocks between each pair of markers, we use this information to generate the predicted whole program basic block path from the sequence of predicted markers. The profiled acyclic paths can be easily identified and counted from this whole program path.

4. EMPIRICAL EVALUATION

To assess the usefulness and effectiveness of our approach, we developed a prototype tool that implements ZOP and performed an empirical evaluation on several software benchmarks. (For simplicity, in this section we use the name ZOP to refer to both the approach and its implementation, unless otherwise stated.) In our evaluation, we investigated the following research questions:

RQ1: How accurate is the profiling information computed by ZOP?
RQ2: How do training input properties affect ZOP's accuracy?

In the rest of this section, we discuss our implementation of ZOP, our evaluation setup, and the results of our evaluation.

4.1 ZOP Implementation

For our evaluation, we used a NIOS II processor on an Altera Cyclone II FPGA. This processor has many of the features of modern complex computer systems (*e.g.,* a 32 bit RISC MIPS-like architecture, a large external DRAM, separate instruction and data caches, dynamic branch prediction, etc.) while also providing features that were extremely useful for developing our understanding of how program execution affects the system's EM emanations (*e.g.,* programmable digital I/O pins, access to programmable logic, and cycle-accurate program tracing). For the evaluation we did not use any FPGA-specific features.

We leveraged LLVM [33] to detect the acyclic paths in the code and to identify instrumentation points and insert instrumentation and used LLVM's C backend to generate instrumented C source code. GCC then compiled this source code to a NIOS binary.

To observe EM emanations, we used a magnetic field probe (a small inductor in series with a tuning capacitor) that was placed directly over the processor's decoupling capacitors. To demodulate and record the signal, we used an Agilent MXA N9020A spectrum analyzer.

Further details of the measurement hardware, software, and ZOP source code and recorded datasets are available at [53–55].

4.2 Evaluation Setup

To answer RQ1 and RQ2, we selected three programs in the SIR repository [45]: replace, print_tokens, and schedule. Table 1 shows

each benchmark's name, its size, the number of markers added during training, and the number of inputs we used during the training and profiling phases.

The decision to use only a few relatively small benchmarks was largely a limitation of the system used and measurement setup. Standard profiling benchmarks are standalone programs designed to run on top of an operating system, but the system used for testing does not have an operating system. To automate measurements, the standalone programs were modified so that each standalone benchmark's main() function was called repeatedly from a wrapper executable. Standalone programs sometimes depend on data memory being initialized to zero when main() is called, and do not clean up before exiting main(). Furthermore, we had to use LLVM's C backend to generate instrumented C code which was recompiled on the target system (in a real application ZOP would use instrumented binaries). These effects introduced some bugs in the benchmarks (*e.g.,* due to differences between the host's and target's treatments of signedness) which were hard to find given the "decompiled" source code and randomly generated inputs. This is why more or larger benchmarks were not measured, though Section 5 presents an argument for ZOP's scalability and future research certainly must demonstrate scalability.

We selected the inputs used for profiling and training carefully to satisfy several criteria. The inputs used for profiling should be selected to provide coverage of all the profiled paths, to demonstrate that ZOP can accurately profile all possible program behaviors with a single training input set. The inputs used for training needed to 1) enable profiling all possible program behaviors (*i.e.,* a single fixed-sized training set is needed that does not require more training inputs as more inputs are profiled), 2) be as small as possible, and 3) be selected in a way that is practical for real applications. Training does not require coverage of all the profiled paths; we found that selecting an input set based on branch coverage (*i.e.,* each branch is observed taken and not taken a few times) worked for ZOP's subpath-matching based path prediction algorithm. Production-quality software should already provide test suites with coverage of this type, and automatically generated unit tests could also provide such coverage.

To satisfy these criteria we started with the existing set of inputs in [45], and for each benchmark randomly split the inputs into two equally-sized disjoint sets, which we call the *training and profiling supersets*. This guarantees the inputs used for training are completely independent of those used for profiling, so there is no risk of interaction between the selection of training and profiling inputs invalidating our results. From the training superset, we randomly selected a minimal subset of inputs that achieved the same branch coverage as the complete set. We then added 150 more inputs randomly selected from the superset to increase the chances of having different acyclic paths covered by different numbers of inputs, so as to be able to study how the characteristics of the training inputs affect ZOP's accuracy and to answer RQ2. We selected 150 as the number of extra inputs based on earlier experiments, as that number of training inputs is not excessively large and yet can provide the variety in coverage that we are seeking. We call this set the *training set*. To determine the set of inputs to profile (the *profiling set*), we randomly selected a subset of the profiling superset that achieved the same acyclic-path coverage as the complete set and then added random inputs to get to 400 inputs, which was the largest number of inputs we could measure in the amount of time we had available.

4.3 Results

To answer RQ1, we first determined the path taken for each profiled input (*i.e.,* the ground truth) by measuring the correct profiling information for each benchmark and each input in the profiling set. Because ZOP estimates profiling at the acyclic-path level, we used the approach in [7] to compute this information. Next, we ran ZOP's Training 1, Training 2, and Profiling phases. For each benchmark and each profiled input, ZOP predicted the number of times each acyclic path was executed and we compared this value with the previously computed ground truth. We then calculated the average accuracy for each benchmark using the following formula:

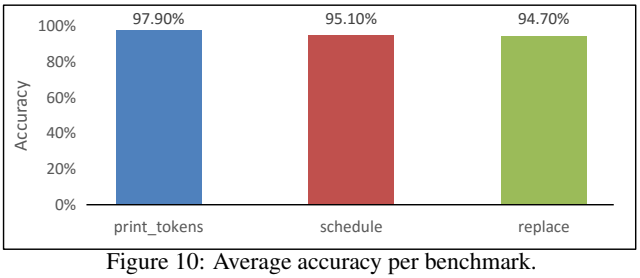

Figure 10: Average accuracy per benchmark.

$$\text{accuracy} = \frac{\sum_{i=1}^{n} g_i a_i}{\sum_{i=1}^{n} g_i}$$

where

$n =$ number of acyclic paths per benchmark.

$g_i =$ actual number executions of acyclic path i(ground truth).

$z_i =$ ZOP (predicted) number of executions of acyclic path i.

$a_i = \min\left(\frac{g_i}{z_i}, \frac{z_i}{g_i}\right) =$ accuracy for acyclic path i.

Therefore, when ZOP underestimates the number of executions of a path, the accuracy is computed as $a_i = \frac{z_i}{g_i}$, whereas when ZOP overestimates the number of executions of a path, the accuracy is computed as $a_i = \frac{g_i}{z_i}$. $a_i = 0$ when $z_i = 0$. To give equal weight to each path execution, each a_i is weighted by g_i.

We show the path profiling accuracy results in Figure 10. On average, ZOP correctly predicts 94.7% of the paths for replace, 97.9% for print_tokens, and 95.1% for schedule. In other words, the profiling information computed by ZOP *without any instrumentation* is always over 94% accurate.

To address RQ2, we computed how the accuracy of ZOP's path count estimates is affected by the number of times each path is exercised by the training set. We show these results in Figure 11. Each data point in this figure represents the accuracy of ZOP's estimate for a single static acyclic path in the indicated benchmark (*i.e.,* a single a_i value). For each benchmark, the figure also shows a fit for a saturating power curve[1] for each benchmark and the curve's goodness of fit (*i.e.,* R^2). We chose this type of curve because, among all simple curves we tried, including linear, quadratic, exponential, etc. it produces (by far) the best goodness-of-fit. A logarithmic scale is used for the x-axis to more directly show the effect of increasing the number of training path instances by an order of magnitude.

For the print_tokens and schedule benchmarks in Figure 11, the accuracy is poor when the acyclic path is executed less than 100 times, but greatly improves beyond this point. This is promising, as it implies that paths can be identified accurately by a relatively small number of inputs that cover them. Moreover, it also implies that accuracy can be improved by adding more training inputs.

The replace benchmark manifests a slightly different behavior. While the accuracy does increase with the number of times the paths are covered during training, there are several paths with more

[1]The curve is $y = a - bx^c$ where x is the number of dynamic instances, y is accuracy, and a, b, and c are constants chosen (for each benchmark separately) to produce the best fit.

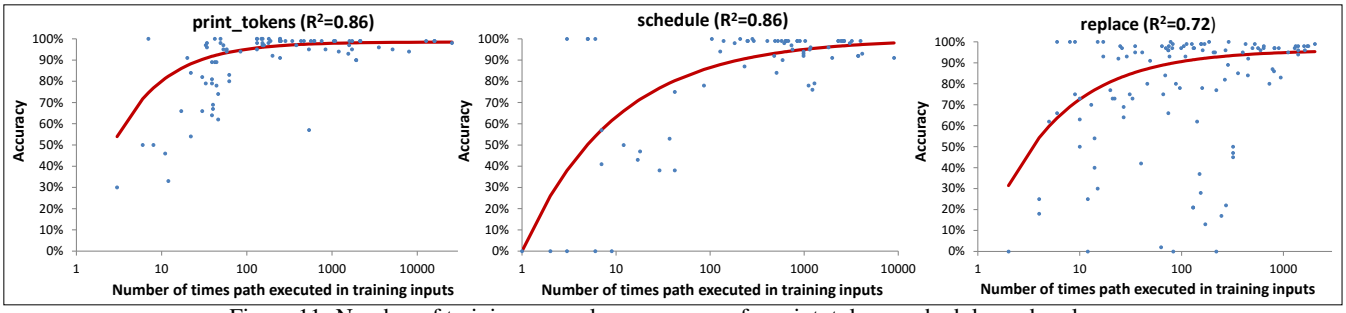

Figure 11: Number of training examples vs accuracy for print_tokens, schedule, and replace.

than 100 training examples that have accuracy below 80%, and even a few below 50%. In general the accuracy for replace does not improve as quickly as for the other two benchmarks as a function of the number of executions of a path during training.

We investigated this difference in behavior and found several possible explanations. First, whereas we expect most paths to have only so many variations in terms of the EM emanations that they can generate (see Section 2), some paths may vary more widely based on the context in which they are executed. Alternatively, some paths may simply have more possible contexts under which they can be executed (*e.g.,* if the structure of the program or some parts thereof contains especially high levels of nesting).

Second, the path prediction algorithm traverses a program's marker graph as shown in the example in Figure 9. This traversal results in the evaluation (and possibly selection) of many impossible paths. The technique that we use to navigate the graph is context sensitive but does not distinguish between different call sites that invoke the same callee from within a procedure. Therefore, the algorithm could reach a callee from a given call site within a procedure and return to a different call site within the same procedure. This is particularly problematic in programs in which this situation occurs frequently and may lead to imprecision and poor predictions.

Finally, for 3 of the 400 inputs in replace's profiling set, the path prediction algorithm got "lost" while exploring the marker graph. As we mentioned while describing our approach, if the waveforms collected during training do not closely match the waveform collected during profiling for more than a short time, the predicted and actual control flows can diverge beyond recovery. Once this happens, the remainder of the prediction for that input is completely incorrect. This condition only happened for the replace benchmark and only for three inputs. This is likely an indication that there is something different about replace and that more training inputs were needed for certain parts of this benchmark. Also in this case, we will perform further investigation to better characterize the peculiarities of replace and use our findings to improve ZOP.

5. THREATS TO VALIDITY

The primary threat to external validity of this work concerns whether or not our results will generalize to larger, more complex programs. Section 4.2 explains why larger programs were not measured. However, an argument can be made that assuming the branch-like coverage used in the evaluation is sufficient for training, and assuming a fixed maximum backtrack depth in the tree search, and assuming a constant number of tree child nodes per parent, ZOP has (1) training time, space, and input requirements linear in program size, and (2) path prediction time linear in program execution time (*i.e.,* constant tree search time per final predicted marker/node).

Results may also not generalize because the programs we selected are not representative of the general program population for reasons different than their size. However, based on our results and on our understanding of the phenomena involved, we do not expect to find dramatic differences in how ZOP worked for these benchmarks and how it would work for different programs. Even if it worked differently in some respect for one of the benchmarks, for instance, its general performance in terms of accuracy was homogeneous across all three programs considered.

Another concern is that using a different processor, having higher clock frequencies, or simply operating in an environment in which the effects of noise are dramatically stronger could considerably affect ZOP's performance. Furthermore, the measurements were performed without an operating system, which makes prediction harder due to multi-tasking and interrupts (the measured system has short interrupts and DRAM refreshes which do interfere with the signal). Previous research has shown that signals similar to the demodulated processor clock used are present in practically every computing system [14], and that removing interrupts from side channel signals [23, 36] is feasible. The evaluated ZOP implementation is a proof-of-concept and numerous opportunities to improve ZOP's capabilities are listed in Section 7.

Finally, threats to construct validity might arise because we used unsuitable metrics to answer our research questions. Given the straightforward nature of the questions we investigated, we believe that our metrics were sensible and appropriate.

In summary, although this work is just a first step in the new direction of EM-based profiling, we believe that the possible threats to the validity of our results are outweighed by our encouraging initial results and by the new possibilities opened by this approach.

6. RELATED WORK

Instrumentation Based Profiling.

Program profiling information is used for code optimization (*e.g.,* [18]), testing and debugging (*e.g.,* [15]), and software maintenance (*e.g.,* [21]). Unfortunately, obtaining code profiles, and in particular path profiles, requires code instrumentation, which is invasive and comes at the cost of high runtime overhead. The path profiling algorithm proposed by Ball and Larus [7], for instance, is an efficient (acyclic) path profiling technique that forms the basis of many other path profilers. This technique was reported to impose an average runtime overhead of 50%, with as much as a 132% overhead in the worst case. Other studies (*e.g.,* [11, 49]) also report similarly high overhead.

A number of techniques have been proposed by researchers to reduce the overhead of profiling. Many of these approaches try to extend or modify the Ball and Larus technique. Selective path profiling techniques (*e.g.,* [4, 11, 35, 49]) aim to reduce the overhead of path profiling by selecting a given set of paths, based on the observation that only a subset of program paths are normally of interest. Targeted path profiling [29] is another related approach

that tries to reduce the execution overhead by not instrumenting the regions in the code where information could be obtained using edge profiling. Pertinent path profiling [8] is yet another technique that addresses the high overhead problem by optimizing the data structures used for profiling. Sampling-based instrumentation approaches (*e.g.,* [6, 47]) use a different approach to reduce the cost of instrumentation and infer profiling information from a sample of runtime events. Finally, partitioned path profiling [1] proposes the idea of parallel path profiling, which profiles a program by evenly distributing the number of probes into multiple cores.

Despite all the work done so far to reduce the runtime overhead of instrumentation based program profiling, profiling still comes at a non-negligible cost in terms of overhead. Although this overhead is tolerable in some cases, it is not always so (*e.g.,* for embedded devices with limited resources or real-time systems). Moreover, instrumentation is an intrusive technique that can change some aspect of a program's dynamic behavior of such code, especially in the case of complex, real-time, and/or multi-threaded systems. Our proposed ZOP system has zero overhead while profiling executions. In return for zero overhead, ZOP requires a training phase and the accuracy is imperfect. This reduction in accuracy may be acceptable for many applications.

Hardware Based Profiling.

Processor hardware features for profiling [5, 27, 28, 44] can never completely eliminate software overhead. Hardware-accelerated profiling will always affect the profiled program due to the need to somehow record profiling information. External hardware tracers and debuggers [34] can profile without software overhead, but require significant processor hardware support to collect and transmit traces off-chip. ZOP has no hardware requirements, which is particularly appealing for applications where any overhead/instrumentation/modification is unacceptable and for systems where hardware profiling support is not available.

EM Side-Channel Emanations.

Electronic circuits within computers generate unintentional yet detectable EM emanations that can be related to program activity [2, 52]. Until recently the study of these emanations focused on two application areas: (1) the potential for these emanations to interfere with wireless communications (*i.e.,* electromagnetic compatibility [25, 41]) and (2) the potential for these emanations to "leak" sensitive data (*i.e.,* accessing a system's vulnerability to EM side channel attacks [3, 26, 30, 48, 51]).

Research interest in EM side channel attacks increased with the introduction mass-market of smartcards (*e.g.,* EMV "chip" credit/-debit cards). Smartcards have processors operating at speeds less than 30 MHz and usually execute a single cryptographic program. EM emanations resulting from this program activity can leak information about embedded cryptographic keys [2, 22]. Smartcard processors have extremely simple architecture and micro-architecture such as 8-bit and 16-bit data widths, no branch prediction, no data or instruction caches, and small on-chip RAM with deterministic single-cycle memory access times.

Recent work has proposed using side channel EM emanations for several new applications such as disassembling a running program based on the EM emanations alone [20,46], instruction profiling for security [38], and also verifying control flow to detect the insertion of malware or other intrusions [9, 37, 39]. These approaches typically focus on identifying individual instructions and do not address predicting control flow through entire realistic programs.

Direct application of such techniques to more complex systems such as desktops and smartphones is difficult because these approaches often require capturing signals at a sampling rate much faster than the devices' clock rate and because the microarchitectural features of complex devices make analysis at instruction and clock cycle scale much more difficult. Despite the difficulties in analyzing complex systems, it has been shown that information can be transmitted from desktops via EM emanations [19], even in the presence of significant countermeasures [52], and cryptographic keys can be extracted using EM side-channel analysis [24]. It has also been shown that some system behaviors can be recognized on long time scales (*e.g.,* web pages can be distinguished [16] and malware can be detected [17], by observing current fluctuations in a power outlet). It has also been shown that differences between instructions can be observed in EM side-channels [13].

The critical difference between ZOP and previous research is that ZOP (1) predicts control flow and execution paths with high accuracy from EM emanations alone with no hardware support, zero software overhead and no interaction with the profiled system, (2) works on more complex programs and more complex systems (*e.g.,* systems with caches and external memory, dynamic branch prediction, higher clock frequencies), (3) relies only on a low bandwidth (625kHz for a 50MHz clock) demodulated signal.

7. CONCLUSIONS AND FUTURE WORK

This paper presented ZOP, a system for zero-overhead profiling which is non-intrusive and requires no hardware modifications or support. In exchange for the ability to profile software without any overhead, ZOP makes a small sacrifice in accuracy ($> 94\%$ accurate compared to a technique based on instrumentation on the benchmarks tested), and requires a training phase.

ZOP uses unintentional EM emanations generated by the profiled system to track a program's execution and to generate profiling information. In ZOP's training phase, the program is instrumented and EM waveforms are recorded while running a set of inputs on both instrumented and uninstrumented code, and the instrumentation records which part of the EM signal corresponds to which part of the code. The profiling phase consists of running the original (uninstrumented and unmodified) program with the inputs to be profiled and recording the system's EM emanations waveforms. The waveforms from training, and their waveform-to-code mapping, are used to predict the execution path taken by the profiled run. Our experimental results show that ZOP can predict profiling information with greater than 94% accuracy for the benchmarks considered in our evaluation.

Future work will demonstrate scalability to larger programs and different devices and investigate opportunities to improve ZOP's accuracy via better algorithms for time warping, training input selection, tree search, usage of program structure/statistics to guide tree search, and waveform matching, as well better probing and noise cancellation. We will also investigate the possibility of using this technology in the context of software security, in particular the usage of EM emanations to recognize anomalies and possible attacks in a completely non-intrusive way.

8. ACKNOWLEDGMENTS

This work has been supported, in part, by NSF grants 1318934 and 1320717, AFOSR grant FA9550-14-1-0223, and DARPA LADS contract FA8650-16-C-7620. The views and findings in this paper are those of the authors and do not necessarily reflect the views of NSF, AFOSR, or DAPRA.

9. REFERENCES

[1] M. Afraz, D. Saha, and A. Kanade. P3: Partitioned path profiling. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 485–495, New York, NY, USA, 2015. ACM.

[2] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM Side—Channel(s). In *Springer LNCS Vol. 2523 - Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45, 2002.

[3] D. Agrawal, B. Archambeult, J. R. Rao, and P. Rohatgi. The EM side-channel(s): attacks and assessment methodologies. In *http://www.research.ibm.com/intsec/emf-paper.ps*, 2002.

[4] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 35–42, New York, NY, USA, 2002. ACM.

[5] Arm cortex-a9 performance monitoring unit. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/BEHEDIHI.html, Accessed 1 May 2016.

[6] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[7] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[8] S. Baswana, S. Roy, and R. Chouhan. Pertinent path profiling: Tracking interactions among relevant statements. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society.

[9] G. T. Becker, D. Strobel, C. Paar, and W. Burleson. Detecting software theft in embedded systems: A side-channel approach. *Information Forensics and Security, IEEE Transactions on*, 7(4):1144–1154, 2012.

[10] E. Biglieri, D. Divsalar, M. K. Simon, P. J. McLane, and J. Griffin. *Introduction to trellis-coded modulation with applications*. Prentice-Hall, Inc., 1991.

[11] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 205–216, Washington, DC, USA, 2005. IEEE Computer Society.

[12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[13] R. Callan, A. Zajic, 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 International Symposium on Microarchitecture (MICRO)*, 2014.

[14] R. Callan, A. Zajic, and M. Prvulovic. FASE: Finding Amplitude-modulated Side-channel Emanations. In *Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 592–603, 2015.

[15] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

[16] S. Clark, H. Mustafa, B. Ransford, J. Sorber, K. Fu, and W. Xu. Current events: Identifying webpages by tapping the electrical outlet. In J. Crampton, S. Jajodia, and K. Mayes, editors, *Computer Security – ESORICS 2013*, volume 8134 of *Lecture Notes in Computer Science*, pages 700–717. Springer Berlin Heidelberg, 2013.

[17] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, K. Fu, and W. Xu. Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *Proceedings of USENIX Workshop on Health Information Technologies*, volume 2013, 2013.

[18] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 95–105, New York, NY, USA, 2002. ACM.

[19] B. Durak. Controlled CPU TEMPEST Emanations, 1999.

[20] T. Eisenbarth, C. Paar, and B. Weghenkel. Building a side channel based disassembler. In *Transactions on computational science X*, pages 78–99. Springer, 2010.

[21] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[22] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: concrete results. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 251–261, 2001.

[23] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer. Ecdh key-extraction via low-bandwidth electromagnetic attacks on pcs. In *Topics in Cryptology-CT-RSA 2016*, pages 219–235. Springer, 2016.

[24] D. Genkin, I. Pipman, and E. Tromer. Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2014.

[25] Henry W. Ott. *Electromagnetic Compatibility Engineering*. Wiley, 2009.

[26] H. J. Highland. Electromagnetic radiation revisited. *Computers and Security*, pages 85–93, Dec. 1986.

[27] Intel performance counter monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor, Accessed 1 May 2016.

[28] Intel vtune amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe, Accessed 1 May 2016.

[29] R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 239–, Washington, DC, USA, 2004. IEEE Computer Society.

[30] M. G. Khun. Compromising emanations: eavesdropping risks of computer displays. *The complete unofficial*

TEMPEST web page: http://www.eskimo.com/~joelm/tempest.html, 2003.

[31] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[32] J. Kraft, A. Wall, and H. Kienle. Trace recording for embedded systems: Lessons learned from five industrial projects. In *Runtime Verification*, pages 315–329. Springer, 2010.

[33] C. Lattner. LLVM. http://llvm.org/.

[34] Lauterbach development tools. http://www.lauterbach.com, Accessed 1 May 2016.

[35] B. Li, L. Wang, and H. Leung. Profiling selected paths with loops. *Science China Information Sciences*, 57(7):1–15, 2014.

[36] J. Longo, E. De Mulder, D. Page, and M. Tunstall. Soc it to em: electromagnetic side-channel attacks on a complex system-on-chip. In *Cryptographic Hardware and Embedded Systems–CHES 2015*, pages 620–640. Springer, 2015.

[37] M. Msgna, K. Markantonakis, and K. Mayes. The b-side of side channel leakage: Control flow security in embedded systems. In *Security and Privacy in Communication Networks*, pages 288–304. Springer, 2013.

[38] M. Msgna, K. Markantonakis, and K. Mayes. Precise instruction-level side channel profiling of embedded processors. In *Information Security Practice and Experience*, pages 129–143. Springer, 2014.

[39] M. Msgna, K. Markantonakis, D. Naccache, and K. Mayes. Verifying software integrity in embedded systems: A side channel approach. In *Constructive Side-Channel Analysis and Secure Design*, pages 261–280. Springer, 2014.

[40] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 378–388. IEEE, 2013.

[41] C. R. Paul. *Introduction to Electromagnetic Compatibility*. Wiley, 2nd edition, 2006.

[42] W. Ruml. *Adaptive tree search*. PhD thesis, Citeseer, 2002.

[43] P. Senin. Dynamic time warping algorithm review. *University of Hawaii*, 2008.

[44] A. Shye, M. Iyer, T. Moseley, D. Hodgdon, D. Fay, V. J. Reddi, and D. Connor. Analysis of path profiling information generated with performance monitoring hardware. In *Interaction between Compilers and Computer Architectures, 2005. INTERACT-9. 9th Annual Workshop on*, pages 34–43. IEEE, 2005.

[45] Software-artifact Infrastructure Repository. http://sir.unl.edu/.

[46] D. Strobel, F. Bache, D. Oswald, F. Schellenberg, and C. Paar. Scandalee: a side-channel-based disassembler using local electromagnetic emanations. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 139–144. EDA Consortium, 2015.

[47] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling, 2000.

[48] W. van Eck. Electromagnetic radiation from video display units: an eavesdropping risk? *Computers and Security*, pages 269–286, Dec. 1985.

[49] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 351–362, New York, NY, USA, 2007. ACM.

[50] R. Wherry. *Contributions to correlational analysis*. Academic Press, 1984.

[51] J. Young. How old is Tempest? *Online response collection, http://cryptome.org/tempest-old.htm*, 2000.

[52] A. Zajic and M. Prvulovic. Experimental Demonstration of Electromagnetic Information Leakage From Modern Processor-Memory Systems. *IEEE Transactions on Electromagnetic Compatibility*, 56(4):885–893, Aug. 2014.

[53] Zero overhead profiling. www.cc.gatech.edu/~orso/software/zop.html.

[54] Zero overhead profiling. http://www.cc.gatech.edu/~milos/zop.

[55] Zero overhead profiling. users.ece.gatech.edu/~alenka/zop.