Schnauzer: Scalable Profiling for Likely Security Bug Sites

William Arthur, Biruk Mammo, Ricardo Rodriguez, Todd Austin and Valeria Bertacco
Advanced Computer Architecture Laboratory, University of Michigan, Ann Arbor
{warthur, birukw, ricardoj, austin, valeria}@umich.edu

Abstract

Software bugs comprise the greatest threat to computer security today. Though enormous effort has been expended on eliminating security exploits, contemporary testing techniques are insufficient to deliver software free of security vulnerabilities. In this paper we propose a novel approach to security vulnerability analysis: dynamic control frontier profiling. Security exploits are often buried in rarely executed code paths hidden just beyond the path space explored by end-users. Therefore, we develop Schnauzer, a distributed sampling technology to discover the dynamic control frontier, which forms the line of demarcation between dynamically executed and unseen paths. This frontier may then be used to direct tools (such as white-box fuzz testers) to attain a level of testing coverage currently unachievable. We further demonstrate that the dynamic control frontier paths are a rich source of security bugs, sensitizing many known security exploits.

1. Introduction

The vast majority of security attacks are enabled by software bugs. Defects which escape detection of software quality assurance can have global impact, such as the Code Red and Sapphire/Slammer worms which utilized buffer overflows for system exploitation. Fueled by these and other high-profile exploits, buffer overflows remain a top security concern [35]. Programs written in popular languages such as C and C++ are a rich source of buffer overflow bugs, as these languages cannot, without high overhead, systematically eliminate buffer overflow vulnerabilities [31]. This then places the burden on test to find potential buffer overflow vulnerabilities before they are exploited.

Commercial software is heavily tested before deployment. Indeed, coding consumes only a small percentage of development effort [27], while studies have shown that testing comprises greater than fifty percent of the cost of software development [4][18]. Regardless, software defects continue to escape detection.

Understanding the way in which latent defects are exploited can reveal critical insight into their prevention. The majority of security-related faults reside in the least likely to be executed code sequences, and by extension, the least tested portions of code [16]. In an effort to heighten initial customer satisfaction, developers tend to focus their limited test resources on the code paths they anticipate users will execute most often, creating significant overlap in developer test and user execution. This in turn shapes a common discovery model used by attackers to locate defects. A malicious user will provide permutations of typical application inputs in an effort to cause slight (but expected) deviations from the well-travelled, and thus well-tested, path of normal execution. Given the combined nature of testing and exploitation discovery models, the location of defects most likely to be exploited can be identified. This exploit-rich code exists just beyond the well-trodden execution paths of testers and users, yet is readily reachable by attackers. We identify these locations as the ***dynamic control frontier (DCF)***.

The dynamic control frontier is a collection of paths rooted in dynamically executed paths. However, these paths are special in that, had the final control decision in these paths executed a different basic block, it would create a new, never-before-seen path. This defines the frontier of the path space executed by an application with respect to a set of inputs. Collectively, the DCF represents the most readily accessible paths of execution which are unlikely to be executed by end-users; consequently, these paths have a high degree of reachability for an attacker. Accordingly, any latent defects in the unexecuted portions of the dynamic control frontier paths are unlikely to be found by users and developers, but these bugs can be quickly uncovered by attackers[1].

It is interesting to look at the dynamic control frontier of an application arising from the test inputs of developers. Indeed we show that this is valuable as we find real vulnerabilities at these locations. However, it is more intriguing to examine the dynamic control frontier for a non-trivial sized population of end users. An attacker is most interested in this frontier as it represents code paths which have not been tested nor executed with any frequency by any user of a particular program. In contrast, any paths frequently executed by users which are not represented in the test suites will probably be devoid of showstopper bugs, as users would otherwise complain. As such, in the construction of a system to profile the DCF, we must be mindful that such a system should analyze the DCF of a large population of users without imposing an unacceptable impact on individual user performance.

CGO'13, 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 © 2013 IEEE...$15.00

[1] Though attempts have been made to quantify software exploitation vulnerability, no known accurate database exists for the quantification of effort required to exploit vulnerabilities [26].

1.1. Contributions of this Work

The goal of this work is not to fix software bugs which drive security exploits; existing tools will be utilized for this purpose. Our goal is to instead show such tools, which often suffer from exponential path explosion, where they can best focus their efforts to find real-world, mission-critical security exploits. This goal merits the work's namesake: schnauzer. Utilized by law enforcement, emergency responders, and medical professionals the schnauzer is a working dog that is exceptionally capable of locating critically important items (illegal drugs, missing persons, etc.). The schnauzer does not actually find the desired item, it instead zeroes in on the locations where its human partner should search -- the perfect metaphor for our work.

The value of the DCF is not to identify code paths with the highest density of bugs. The value of the DCF is to identify the code paths which are least tested by developers and users, while also most readily accessible to attackers. We will demonstrate that there is mounting evidence that bugs hidden within the DCF are more likely to be exploited, and therefore are of the greatest merit to discover.

In this work, we develop a low-overhead and efficient software mechanism that effectively identifies the dynamic control frontier over a large population of users of an application. The approach utilizes a distributed sampling method in which individual user machines locate dynamic control frontier paths and occasionally communicate them back to the developers. Developers utilize white-box testing and dynamic program analysis to fully test these paths for security vulnerabilities. Over time, the system raises the bar on what constitutes the DCF, and thus raises the bar on the difficulty of finding security vulnerabilities. In this work, we make the following novel contributions:

- We present an effective, scalable, and decentralized approach to identifying the dynamic control frontier for a program running across a large population of users.
- We present a software implementation for harvesting dynamic control frontier information from individual user machines. The approach utilizes dynamic code instrumentation to limit the impact to application execution while providing appropriate coverage of the dynamic control frontier in the aggregation of users.
- We demonstrate the value of the dynamic control frontier by showing that many known security vulnerabilities may be found there. We show that dynamic control frontier paths sensitize known exploits identified by the NIST National Vulnerabilities Database.
- We evaluate the effectiveness of the approach by exploring the performance—cost tradeoffs while harvesting DCF paths. We also developed a novel whole-path analysis technique that allows us to gauge the coverage of the approach (i.e., the total percent of dynamic control frontier paths found as a function of total population run time). We present results for a wide range of non-trivial software packages that show our approach achieves good coverage while keeping performance impacts low.

The remainder of this paper is organized as follows. Section 2 provides an in-depth overview of the dynamic control frontier. Section 3 details our DCF profiler, Schnauzer. Experiments conducted to evaluate the benefits and costs of DCF profiling, and a full analysis of the results are delivered in Section 4. Finally, Section 5 lists related works while Section 6 gives conclusions and future work.

2. Dynamic Control Frontier Discovery

Security exploits arise from bugs which escape detection by the developer. Often, hidden bugs only appear when sensitized by the proper path [4]. For example, attempting to free a pointer after already doing so previously (double free). The predominance of path-sensitized bugs follows from the observation that commercial software generally achieves both branch and code coverage but remains deficient with respect to path coverage. Unfortunately, achieving path coverage is currently an intractable problem for applications of any appreciable size. This is due to the explosion in the number of paths, ultimately limiting path testing to a tiny subset [25].

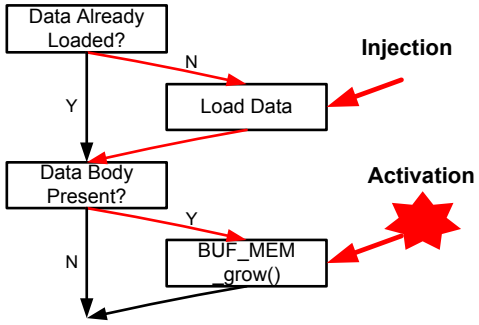

Figure 1 - Path Sensitization. Represented is the high-level overview of a security bug from the National Vulnerability Database (CVE-2012-2110 and CVE-2012-2131) for the *OpenSSL* application. In experimentation, this vulnerability was found to be sensitized by the dynamic control frontier. Here, a buffer overflow attack results from crafted data of an RSA public key. Note that the vulnerability is only sensitized by a single path (N, Y), indicated in red.

Given the combination of path explosion and the need for path sensitization to activate bugs, it is inherent that exhaustive testing to locate bugs is an infeasible approach [34]. Thus, software testers are forced to constrain the path space to some feasible subset [4]. The quandary of test allocation, or the optimal test resource allocation problem (OTRAP) [25], is generally approached from the perspective of software reliability and cost [15][32], rather than security. Identifying the subset of paths which are likely to contain bugs, which are in turn likely to be exploited, would yield the highest productivity in test relevant to potential exploit detection.

This subset of paths, deemed highly likely to result in exploits, is encompassed by the dynamic control frontier. The dynamic control frontier represents the border of dynamic execution between dynamically seen paths and those which are unseen. The first unexecuted basic block of these dynamic control frontier paths represents a location that is likely to hide a security exploit.

Consider the known security bug modeled in Figure 1, a high-level representation of an exploit discovered in *OpenSSL*. The bug, documented in the National Vulnerability Database [24] and which enables a buffer overflow attack, is only sensitized by a specific path of execution. When handling DER encoded data, maliciously crafted data can activate the vulnerability. Note that the (N, N) (Y, N) and (Y, Y) paths were seen with some frequency while running *OpenSSL ssltest* and do not sensitize the bug, but the path (N, Y) was not seen, and hence represents the DCF where the bug was sensitized by the untested path.

Paths which remain unexecuted, that are not comprehensively tested, will continue to harbor latent bugs. This space, all un-executed paths, is still far too large for comprehensive testing. However, the code paths which are immediately outside the dynamic execution are the ones which are the most readily reachable by attackers. Because the dynamic control frontier is unlikely to ever be executed, the security bugs in this code will typically only be fixed when an active exploit is exposed. Debugging the DCF will force any attacker to probe deeper into the code. This will raise the bar in terms of the amount of effort required to attack programs, and make it much more difficult for an attacker to find good security bugs. In Section 4 we show that the number of dynamic control frontier paths is relatively few and quite rich in security exploits.

$$DCF(P) = \{p_i, p_j, \dots\ p_m\}$$
$$p_i = \langle bb_1, bb_2, \dots, bb_{n-1}, bb_n\rangle \mid$$
$$\langle bb_1, \dots, bb_{n-1}\rangle \in EX(P)$$
$$\wedge \langle bb_1, \dots, bb_{n-1}, bb_n\rangle \notin EX(P)$$
$$EX(P) = \{\ \dots all\ paths\ executed\ \dots\}$$

Figure 2 – Dynamic Control Frontier. The dynamic control frontier of an application P, DCF(P), is the set of paths *p*, comprised of a series of *n* basic blocks. These paths consist of basic blocks, where the first *n-1* basic blocks were in fact executed in EX(P), but the whole series of *n* was not.

As defined in Figure 2, the dynamic control frontier of an application are the sets of length-*n* paths, comprised of basic blocks, where the first *n-1* blocks have been seen to be executed, but the full series of *n* basic blocks has not been seen to execute. Thus, the dynamic control frontier is a path in which the last control decision to basic block bb_n creates a never-before-seen path of execution. Basic block bb_n is the likely site of a security exploit, sensitized by the path leading to it (bb_1,..., bb_{n-1}). More formally, the dynamic control frontier is defined as follows. The dynamic control frontier DCF(P), of an application P, is the set of paths *p* comprised of a series of *n* basic blocks. These paths consisting of *n* basic blocks, where the first *n-1* basic blocks form a path (of length *n-1*) in the set of executed paths, EX(P), but the full length-*n* path of basic blocks is not contained in the set of executed paths EX(P).

2.1. Computing the Dynamic Control Frontier

Determining the exact dynamic control frontier, which we call the ***ground truth DCF***, for a given application execution can be accomplished by analyzing its execution trace.

The ground truth DCF computation method is given in Figure 3. First, a trace of basic blocks is collected for an application in execution with respect to a set of inputs. This trace is then scanned for all length-*n* paths of basic blocks. These paths are sorted into sets by their length-*n-1* path prefixes. For each path within the set of paths with common path prefixes, if there exists any control exit from the *n-1* block (the last block of the path prefix) which is not represented in the set, then this path prefix, along with the unseen control exit block, is a member of the set of ground truth DCF paths.

$$EX(P) = \{\ \dots all\ paths\ executed\ \dots\}$$
$$GTDCF(EX(P)) = \{\ \dots all\ ground\ truth\ DCF\ paths\ \dots\ \}$$
$$p_x = \langle bb_1, bb_2, \dots, bb_{n-1}, bb_n\rangle$$
$$pp_y = \langle bb_1, bb_2, \dots, bb_{n-1}\rangle$$
$$PREFIX(p_x) = \langle bb_1, bb_2, \dots, bb_{n-1}\rangle \in\ p_x$$
$$EQ(p_x, pp_y) = PREFIX(p_x) \leftrightarrow\ pp_y$$
$$\big(\ \forall pp_s \in EX(P)\big)\big[\ \exists p_t\big[EQ(p_t, pp_s) \wedge\ p_t \notin EX(P)\big]$$
$$\rightarrow\ p_t \in GTDCF\big(EX(P)\big)\big]\}$$

Figure 3 - Ground Truth Dynamic Control Frontier. The ground truth DCF of an execution instance EX(P) of application P, GTDCF(EX(P)), is a set of paths *p*, comprised of a series of *n* basic blocks. These ground truth paths are those where their length-(*n-1)* path prefix was executed, EX(P), but their entire length-*n* paths were not.

2.2. Profiling the Dynamic Control Frontier

Establishing the ground-truth set of dynamic control frontier paths, needed to provide good coverage of the dynamic control frontier paths for an application, is prohibitively expensive to do widely. Analyzing an execution trace assumes a finite application run. Also, such a trace grows to unmanageable size after long execution periods. For example, collecting a trace consisting of purely conditional branch information, limited to instruction address and branch direction, while executing the *SQLite* test suite *quick test* accrues over 300 GB of data during ground truth analysis of a 154 billion instruction length execution. Further, keeping track of all potential DCF paths during a program's execution is a significant performance overhead; for example, the ground truth DCF analysis of *SQLite* using *Pin*-based instrumentation [21] resulted in an average application slowdown of 26X.

Thus, a practical method must be developed to profile an application for the ground truth DCF. This can be achieved by sampling a small subset of the paths executed by an individual user and combining these samples over a large user population. While observing the execution of an application, at occasional intervals, a path is selected for profiling. A hypothesis is made from the length-*(n-1)* prefix seen in execution and the length-*n* path derived from this prefix which is not seen (i.e., the hypothesis is constructed by taking the opposite branch direction out of the last control decision seen to execute). We then hold this

hypothesis for an extended period of time, waiting to see if the path is executed, and thus the hypothesis refuted. If the hypothesis path is not seen to be executed for this holding period, it is considered a good candidate for a DCF path. If, however, the hypothesis is seen to be executed, the hypothesis is refuted and not considered further.

The dynamic control frontier can be established for any single execution of an application. However, this frontier will vary depending on the inputs to an application for a given instance. Consequently, the DCF discovered for a single user is of limited value. A user may run the application with inputs which ultimately refute a hypothesis considered a good candidate by another user. Potential DCF candidates are therefore collected into a single ***global path filter*** database which is shared with all users over time.

Initially, dynamic control frontier path hypotheses will be sampled by multiple users. If a path is refuted, it will be removed from the global path filter. Otherwise, as hypotheses in the global path filter age they come to represent true dynamic control frontier paths. These venerable DCFs can then be used to filter hypothesis creation on individual hosts as profiling these high-confidence DCF paths would provide no benefit. Figure 4 depicts an overview of DCF sampling.

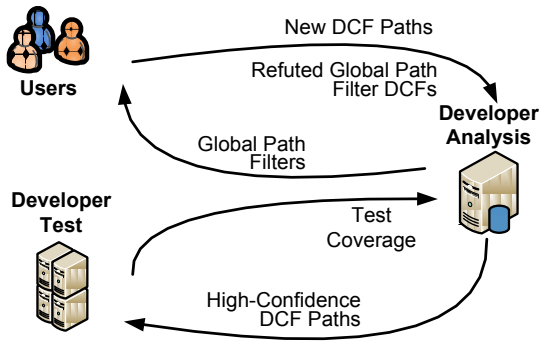

Figure 4 - Dynamic Control Frontier Sampling System. Users profile application execution while sampling to discover dynamic control frontier paths. These DCFs drive developer analysis, which directs testing methods. Global path filters coordinate work between users.

2.3. Leveraging the Dynamic Control Frontier

Once dynamic control frontier paths begin to materialize, they must be harnessed to find security vulnerabilities. We can use white-box testing to deeply analyze DCF paths for security vulnerabilities. ***White-box testing*** has emerged as an effective testing approach to overcome this limitation [12].White-box testing is designed to fully explore the dynamic control flow within a code module. The approach essentially is the reverse of fuzz testing. Rather than buffeting the code with random inputs in the hope of exposing new code paths, the approach instead selects a specific code path for testing and then uses SAT-based tools to deduce the inputs to the program or function that would cause the path to execute.

White-box testing has offered the ability to improve fuzz testing by a considerable margin [11], however, the approach still has limitations. For any non-trivial program, the number of paths that must be explored by white-box testing quickly overwhelms the computational capability of existing tools. For example, if the code in Figure 1 is embedded within a loop, the number of paths will be exponential with the number of loop iterations, e.g., at 1000 iterations the number of unique paths is 2^{1000}.

DCFs have the potential to become a divining rod for white-box testing tools, showing them where to spend their efforts to search for vulnerabilities. The DCF instructs which path to follow to reach the likely bug site; the SAT engine typically found in white-box testing tools can determine the inputs necessary to execute the DCF path (or determine that it is an infeasible path of execution). It is interesting to note that a key insight from white-box testing is that bugs are not far from the path of execution, they are just out of reach. Dynamic control frontier profiling leverages this same insight by identifying code just beyond the demarcation of executed code. To effectively expose bugs, attackers must explore code that is not executed ***by any user***. As such, ***there is much promise to improve security vulnerability analysis via white-box testing by identifying dynamic control frontier code paths over a large population of users' machines***.

3. Schnauzer: A Distributed DCF Profiler

To validate our distributed approach to profiling the dynamic control frontier, Schnauzer was built. Our profiler was implemented as a client tool utilizing the *DynamoRIO* dynamic instrumentation tool platform [3]. The goal in developing Schnauzer was to push DCF analysis into the user space by using sampling to demonstrate the potential to minimize runtime overheads associated with DCF profiling, all the while achieving coverage of the ground truth DCF.

Efficiency is critical when profiling in the user space. Thus, profiling at the abstract level of the basic block is undesirable. As such, the conditional branches of an application are preferred to model paths for the dynamic control frontier. Conditional branches are chosen as they may be observed directly from the execution stream, unlike basic blocks. Furthermore, they provide an elegant representation of the control flow of an application, reducing the amount of information necessary when compared to basic block analysis. ***A dynamic control frontier path is simply then a path derived from a length-n executed path of conditional branches in which the trailing conditional branch only goes one direction*** – in this case the length-*n* DCF path is the same path that exits in the opposite (and yet unseen) direction.

The *DynamoRIO* implementation of Schnauzer, shown in Figure 5, works as follows. An application begins unmodified execution through *DynamoRIO*. At random, bounded sampling intervals, the next *n* conditional branch edges are instrumented. At each branch edge seen during execution, a few assembly-level path tracking instructions are added, as shown in Figure 6, to record the occurrence of the path to a memory location when the edge is re-executed. Upon reaching the last branch in the length-*n* path, the

unseen edge of this last branch becomes a potential node on the dynamic control frontier. A function call, referred to as the refuting instrumentation, is inserted at this edge which, when called, invokes a routine in our *DynamoRIO* client to evaluate the path leading to the edge. This path/node combination constitutes a DCF hypothesis, as it has not yet been seen during execution and the path formulation information for this hypothesis is recorded.

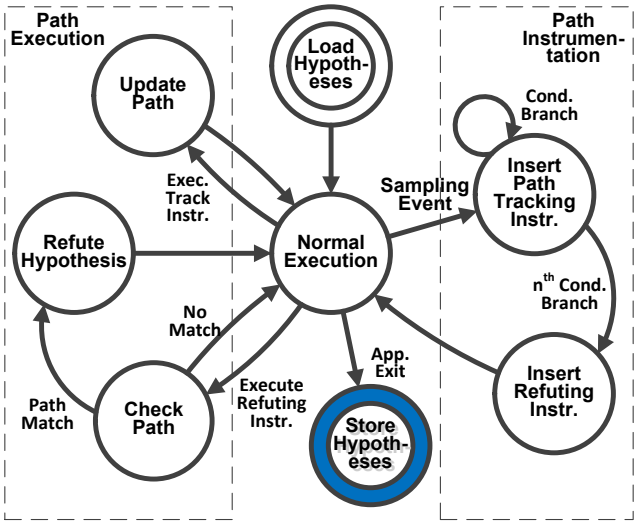

Figure 5 - *DynamoRIO* DCF Profiling Client. The application executes unmodified until a path is selected for profiling, at which time lightweight instrumentation is added only to the selected path. This instrumentation updates the path history when executed. In the event the last edge of a hypothesis is executed (the refutation instrumentation), the path history is checked for a match. Hypotheses may persist across application executions.

The application then continues to execute uninterrupted. If at any time the refutation instrumentation call is invoked (i.e., the previously unseen branch edge from the last branch in the hypothesis is taken), the function will compare the recorded incoming path to the hypothesis' path prefix to determine if the path leading up to the edge matches that of the current hypothesis. If the dynamic path matches the hypothesis, then that hypothesis is refuted, and the *DynamoRIO* code cache is flushed to remove the potential DCF hypothesis. By only instrumenting paths which are hypothesized to be DCF paths, overheads remain low.

If after some long period of aging time a hypothesis has not been seen in the execution trace, this hypothesis is considered confirmed. At that time it is added to the set of dynamic control frontier hypotheses, which will be reported en masse to the developers at a later time. Before any new hypothesis is formed, it is checked against the global path filter plus the internal list of recently recorded DCFs to avoid duplication of effort. Our client also loads and stores hypothesis and global path filter state whenever profiling is invoked. Accordingly, profiling persists across an arbitrary number of application executions. The work of confirming hypotheses, as well as initiation of random sampling, is performed by a separate thread of execution created within DynamoRIO. This allows such work to be completed without slowing the target application.

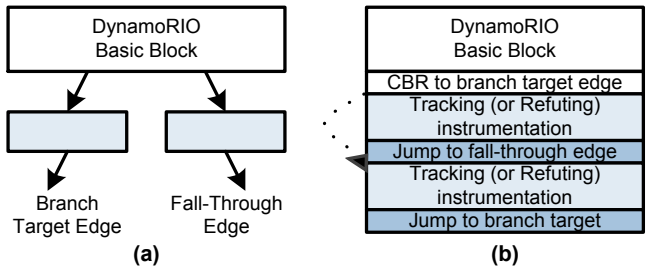

Figure 6 - *DynamoRIO* DCF Profiling Dynamic Instrumentation. DynamoRIO basic blocks **(a)** are instrumented with assembly-level instructions inserted only on branch edges for paths being actively sampled. **(b)** Shows the layout of new basic blocks with instrumentation. Note that for a single active hypothesis, only the relevant subset (tracking *or* refuting) would occur, and only on a single edge for each conditional branch in the hypothesis.

4. Experimental Evaluation and Results

To fully understand the benefit of the DCF, both the cost and accuracy of DCF profiling were evaluated.

4.1. Benchmark Applications

Benchmarks were carefully selected to represent commonly used programs. These programs are popular, network facing applications which increases their profile to attack. Additionally, we sought out programs that had access to high-quality test suites, especially fuzz testers, such that DCF path profiling could run for extended periods of time to locate the code that developers (knowingly or not) chose not to test. The *OpenSSL* (1.0.1c) toolkit, *Python* interpreter (2.7.1), *Tor* (The Onion Router 0.2.2.37), *InspIRCd* Internet Relay Chat server (1.1 and 2.0), and *Pidgin*(2.10.4) executed the regression test suites with their respective distributions. The *SQLite* (3.7.7) benchmark was executed with the fuzz testing components of the standard tcl test library. The *tshark* network analysis tool (1.6.0) was tested with the fuzz test generation tool included with the *tshark* distribution.

4.2. Experimental Framework

The testing platform consists of 64-bit x86 servers running *Ubuntu* 11.04 *Natty Narwhal* with *Linux* kernel 2.6.38-10-generic. All path information was gathered using either the *DynamoRIO* [3] or the *Pin* [21] binary instrumentation tools to instrument benchmark applications. There are four major variables relevant to DCF profiling; path length-*n*, sampling interval, hypothesis age threshold, and the number of concurrent hypotheses for a given analysis. Of these, path length has a direct relationship with the DCF, while the other three are sampling parameters.

Since bugs are often sensitized by a particular path, the DCF has an important relationship with path length. The bug represented in Figure 1 would not be sensitized by a path length of 1 (branch coverage), as all branches involved see both edges in normal execution. This yields no ground truth DCF paths, as described by Figure 3, and the bug

would therefore escape detection by DCF profiling. This observation motivates the desire for longer DCF paths. However, as path length grows, the odds of the same path executing again reduces, potentially resulting in the DCF becoming the set of all paths. To determine the optimal path length for DCF profiling, the relationship between path length and known security defects was explored. This analysis, shown in Section 4.6, determined that a path length of 4 was most effective. For this reason, the subsequent experiments were conducted with a path length of 4 conditional branches.

To further reduce the runtime overhead due to instrumentation, long intervals of time can elapse between hypothesis formulation and the aging threshold. In all overhead and coverage experimental results shown, the sampled hypothesis formulation period is randomly distributed between 1 and 100 milliseconds of instrumented program run time while the hypothesis aging period is 10 seconds. These values were found to facilitate an effective coverage rate while maintaining accuracy of profiled dynamic control frontier paths with respect to the ground truth DCF.

The number of concurrent path hypotheses is limited to a single hypothesis. While imposing the lowest overhead, a single hypothesis also limits sampling capacity. Later in Section 4.5, we show that a single hypothesis is virtually as effective as multiple concurrent hypotheses in establishing coverage of the DCF ground truth.

4.3. Ground Truth Dynamic Control Frontier

A custom pintool was created to perform whole-path analysis of a program to discover all of the dynamic control frontier paths. The whole-path analyzer generates the entire conditional branch trace for all of the program's test inputs. We then scan this trace for all unique length-*n* paths, and then rescan the trace to determine which of the discovered paths exit in only one direction. The opposite exit of the paths' prefix constitutes the complete set of DCF paths that our sampling system could discover, and these paths form the ground truth necessary to gauge coverage of the proposed sampling mechanism. Table 1 shows the application trace and ground truth DCF set size for all benchmarks. The number of ground truth DCF paths is seen to be very few when compared to the potential path space arising from the large execution traces.

To assess the reduction in path space, we statically analyzed the potential number of length-*n* paths which could be executed for an application. A conservative estimate was made based on extending the cyclomatic complexity measure (CCM) [22] to include inter-procedure paths. Developed by McCabe, CCM is a simple metric to assess path complexity for a function. Leveraging CCM, we estimated the number of length-*n* paths within a given function. We then extended this to inter-procedure paths by identifying the length-n paths which may extend beyond the function, both leading into and exiting from the function, for all call sites within the code base. This measure, though an estimate, is considered quite conservative as it does not consider the path space expansion arising from loops. This inter-procedure complexity measure adapted from CCM is shown in the third column of Table 2 for all benchmarks.

Application	# Instructions Profiled	# Potential Length-*n* Paths	# Ground Truth DCF Paths
SQLite	16,948,864,926	13,642,304	17,351
OpenSSL	5,014,034,838	23,221,696	10,086
tshark	684,000,546	38,467,136	178
Python	656,068,272	12,175,712	35,206
Tor	118,310,256	1,191,280	10,639
InspIRCd	46,246,206	11,165,696	3,950
Pidgin	4,762,914	6,833,360	3,641

Table 1 - Benchmark Applications. Profiled instruction trace size for ground truth analysis is shown in the second column. The third column represents the potential number of length-4 paths, measured from an inter-procedure cyclomatic complexity measure. The final column shows the number of length-4 DCF paths within the profile trace.

4.4. Analysis of DCF Sampling

We evaluated the runtime overhead from profiling with Schnauzer as well as the accuracy of the coverage with respect to the ground truth DCF. Figure 7 details the runtime overhead experienced when profiling applications with our *DynamoRIO* client.

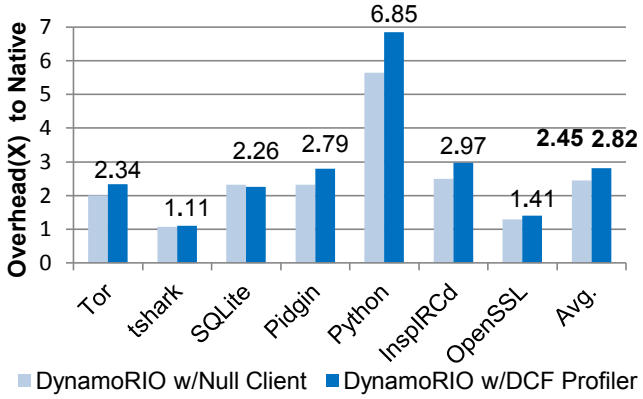

Figure 7 - Sampling Overhead. Runtime overheads for applications are minimally above the slowdown experienced from the *DynamoRIO* core with a NULL client.

In all cases the majority of execution slowdown (2.82X average application runtime overhead) is attributed to the *DynamoRIO* core, which averages 2.45X runtime performance penalty compared to native execution. This small Schnauzer instrumentation overhead is due to a lightweight approach of only instrumenting code paths which are being actively profiled, which results in a 15% overall increase in execution time relative to *DynamoRIO* with a NULL client. The slight improvement in overhead experienced by *SQLite* from our client is attributed to the alteration of fundamental *DynamoRIO* operating mechanisms (e.g., code cache) which affects performance, in this case positively.

Given the general-purpose nature and powerful flexibility of *DynamoRIO*, a lighter-weight DCF path-specific dynamic

instrumentation tool could potentially significantly improve DCF profiling performance. Indeed custom tools have been shown to be highly effective when compared to binary instrumentation platforms like *DynamoRIO* and *Pin*. Zhao et al. demonstrate a low-overhead tool for shadow memory translation with *Umbra* [33], while Bosman et al. develop a dynamic taint analysis tool, *Minemu* [2], which is significantly faster than any competing general-purpose solution. Minemu demonstrated that, for such dynamic analyses, slowdown was not a fundamental property but instead arose from non-specialized implementations.

DynamoRIO was chosen as an initial development platform for power and flexibility combined with rapid accurate prototyping of DCF profiling. Although runtime overheads demonstrated generally remain higher than desired, we believe initial deployment is certainly possible (and planned) with the current framework for a range of applications.

Because we locate DCFs with sampling, there is legitimate concern as to whether or not the technique will observe all of the possible (ground truth) DCFs, and moreover, will all of the DCFs be identified in a reasonable amount of run time. As shown in Figure 8, our profiler locates the vast majority of DCFs in a short period of time. Larger applications, with billions of instructions, necessitate trillions of instructions of execution to receive good profiling coverage of all possible DCFs. This translates to at most ten thousand users profiling the application a single time each, certainly within reach of a modest user population.

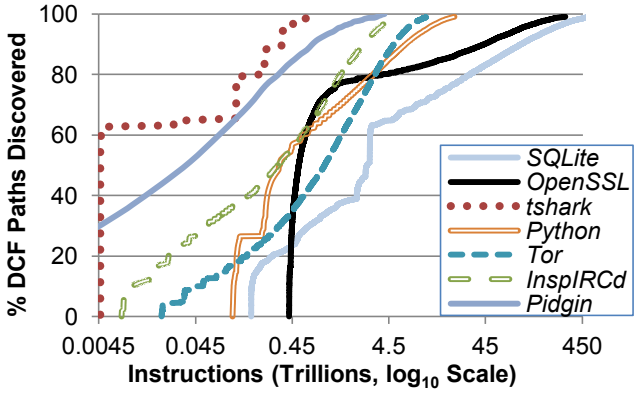

Figure 8 - Path Coverage via Sampling. All benchmarks attain 100% dynamic control frontier path coverage. Even application traces of billions of instructions achieve coverage within trillions of instructions. Thus, a user population of less than ten thousand can profile a trace in a single run. Profiling is done while utilizing only a single active hypothesis at any time, and with a path length of 4 conditional branches.

Additionally, because sampling may deem a path a DCF, which in fact both directions were executed (but only one was observed), the accuracy of sampling must also be measured. Figure 9 shows the accuracy with which DCF paths are selected while profiling. Accuracy is given as the percentage of likely DCF paths, discovered by sampling, which are in the set of ground truth DCF paths for the application trace. Some applications achieve perfect accuracy while sampling, and overall Schnauzer is almost 99% accurate in profiled DCF paths with respect to the ground truth DCF.

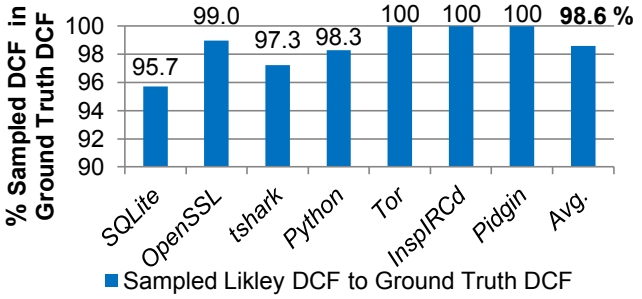

Figure 9 - Sampling Accuracy. The percentage of likely DCF paths discovered by sampling which appear in the set of ground truth DCF paths.

4.5. Schnauzer Profiling Scalability

The dynamic control frontier is most valuable when it is derived from a sizeable population of end-users. Further, it is expected to profile an application for its entire life cycle. Schnauzer must therefore scale with application size, duration of execution, and population of users.

As shown in Table 1, the number of DCF paths for an application is quite small when compared to the potential path space of such a long execution trace, greatly narrowing the domain for test. It must be considered, however, to what extent the dynamic control frontier path space will grow as an application execution continues unbounded. Figure 10 demonstrates that as trace length grows ever larger, the ground truth DCF path space grows linearly. This gives confidence that the path space for test, the number of ground truth paths which must be discovered while sampling, and the incidental work such as updating the global path filter, will all remain within a bounded, manageable range.

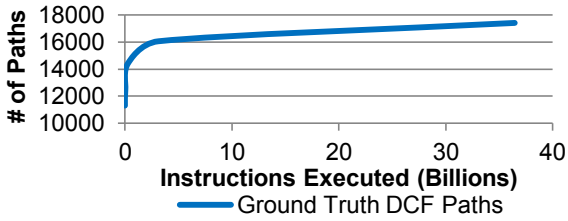

Figure 10 - DCF Path Growth. As the number of executed instructions grows, ground truth DCF path space remains small. The application shown is *SQLite*, executing increasing durations of the fuzz testing component of the test suite.

Schnauzer scales very well with increasing path length. As shown in Figure 11, to facilitate the highest degree of path sensitivity, ***path length has no appreciable effect on sampling overhead for paths ranging from 1 to 64 conditional branches.*** This is due to the lightweight approach for path instrumentation, as only a few assembly-level instructions are added to the path. As well, the number of DCF paths will increase linearly with path length. Given this, paths of up to a length of 64 conditional branches may be analyzed with little impact to performance, should the need for greater path sensitivity arise.

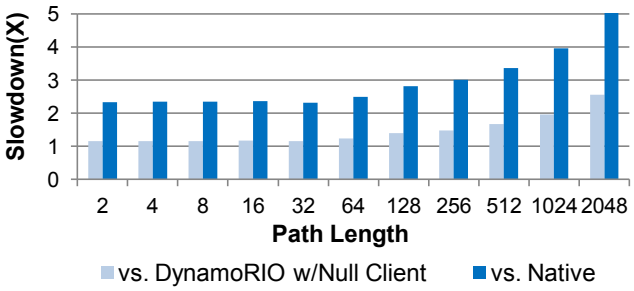

Figure 11 - Path Length Scaling. As the profiled path length increases in Schnauzer, the performance overhead rises slowly. For paths up to 64 conditional branches, little difference is seen. The benchmark shown is *Tor*.

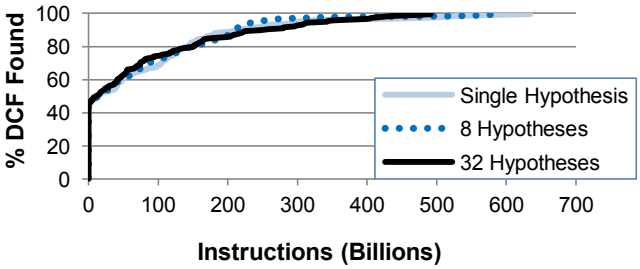

Figure 12 - Concurrent Hypotheses. The number of active hypotheses has minimal impact on sampling coverage. This is due to the inclusion of the global path filter and local sampled path list to eliminate redundant sampling. Benchmark shown is *tshark*.

Profiling overheads are kept low by limiting sampling frequency and the number of paths concurrently being sampled. Figure 12 reveals ***only a single path need be actively profiled at any time***. The utilization of a global path filter and local list of recently sampled DCF paths eliminates redundant work and allows all DCF paths to be discovered in an acceptably similar amount of time, regardless of the number of concurrent hypotheses.

Scalability at the system level is achieved as well. Given the rate of dynamic control frontier path discovery while profiling *SQLite*, the overall bandwidth requirement from a population of users to the aggregation point at the developer is under 5 bytes/second per user. Such a result suggests that a single central server shard could likely serve 10,000's of individual user machines performing DCF path profiling. As the path space of an application is explored, the influx of new path information will decrease. To enhance profiling over the entire life cycle of an application, the aging time for a DCF hypothesis can be increased. Increasing this age threshold brings profiled DCF paths closer to the ground-truth set of DCF paths for the entire lifecycle of an application.

4.6. DCF Correlation with Real Vulnerabilities

It has been shown that a large execution trace can contain a tractable number of dynamic control frontier paths for comprehensive test. However, it is necessary to demonstrate that this information delineating the frontier of dynamic execution is also a fertile source of real security exploits. To establish the relationship between the dynamic control frontier and security exploits, we sought to find if profiled DCF paths indeed sensitized important security bugs. The DCF paths gleaned from ground-truth analysis were compared to bug reports from fixed security bugs. Fixed bugs were chosen so as to know the precise location of an exploited bug within the source code. These bug locations could then be compared to the profiled DCF paths. If the location of a known bug is found to be sensitized by and located directly at the end of a DCF path, then the bug can be said to have been effectively hidden behind the dynamic control frontier.

Application	Vulnerability	Security Advisory
OpenSSL	Buffer Overflow	CVE-2012-2110
	Buffer Overflow	CVE-2012-2131
	Integer Underflow	CVE-2012-2333
SQLite	Buffer Overflow	CVE-2007-1888
Tor	DoS	CVE-2011-0492
	Buffer Overflow	CVE-2011-1924
Pidgin	DoS	CVE-2011-4939
tshark	Format String	CVE-2009-0601
	DoS	CVE-2011-0538
	DoS	CVE-2012-2394
Python	DoS	CVE-2010-2089
	DoS	CVE-2012-2135
InspIRCd	Buffer Overflow	CVE-2008-1925
	Heap Overflow	CVE-2012-1836

Table 2 - Software Vulnerabilities Sensitized by Dynamic Control Frontier Paths. Known software vulnerabilities identified in the NIST National Vulnerabilities Database (NVD) were shown to be sensitized by DCF paths.

As seen in Table 2, known security bugs are sensitized by the dynamic control frontier. ***A total of 14 security exploits were found at the dynamic control frontier*** for the profiled benchmark applications. The security exploits are drawn from the National Vulnerability Database (NVD) [24], which is maintained by the National Institute of Standards and Technology (NIST). The database was searched for Common Vulnerability Exposures (CVE's) [23] existing in benchmark applications. Not all vulnerabilities listed in the NVD for our benchmark applications were sensitized by DCF paths. Some, such as configuration errors, are beyond the scope of DCF path analysis. Others were simply not sensitized by the set of DCF paths profiled from our test inputs. However, these results are a strong affirmation that the control frontier indeed harbors bugs which are likely to be exploited.

It is interesting to note that profiling the dynamic control frontier is not only fruitful for finding security bugs. We also have early evidence that it is a prime target to search for software bugs in general. To this end, a separate analysis of the *SQLite* application was performed. In this analysis the ground-truth DCF was compared to the most recently fixed bugs in the SQLite code base. We found that ***12 of the most recent 20 bugs fixed in SQLite lay on code paths sensitized by the dynamic control frontier.*** Of those 12 bugs, 5 were clearly enabling security vulnerabilities.

To determine an optimal path length for our experiments, the benchmarks were profiled for DCF paths of varying length, as shown in Figure 13. These sets of DCF paths were then analyzed to determine which vulnerabilities, listed in Table 2, would be sensitized by the set of DCF paths for a given path length. Within the scope of our experiments, the number of DCF paths increases roughly linearly with path length. More vulnerabilities are identified by the growing set of DCF paths. All vulnerabilities shown in Table 2 were discovered with a path length of 4 branches, with no other CVE entries indicated by longer paths. Therefore, this path length was selected our experiments. This coincides with the observation that bugs may be more likely to be found with shallow control flow activation rather than being correlative with path coverage [12]. It is important to note that even in the event that this path length is not optimal for another application, Schnauzer is amenable to longer paths as well.

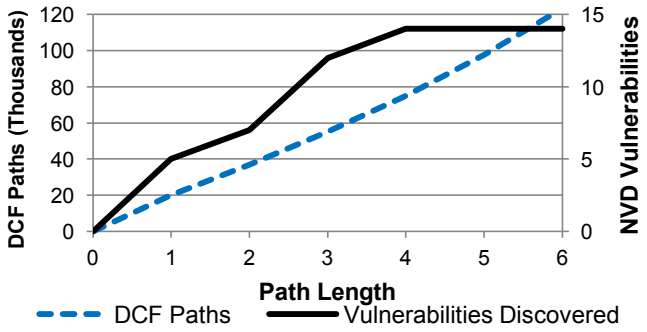

Figure 13 - Impact of DCF Path Length on Vulnerability Discovery. Shown is the relationship between path length and vulnerabilities discovered, for the sum of all benchmarks listed in Table 1 and vulnerabilities identified in Table 2. Benchmarks were profiled for paths of varying lengths. As path length increases, the number of DCF paths increases, with more bugs sensitized. All vulnerabilities in Table 2 are discovered by the set of DCF paths profiled for a path length of 4.

5. Related Work

Much work has been done in the pursuit to identify and fix security vulnerabilities. Even more effort has been expended to deliver comprehensive testing of applications. Some related works are entirely complementary to Schnauzer. Other efforts assist in building a foundation for finding vulnerabilities but are not entirely sufficient themselves to accomplish the central goal of identifying code paths likely to be exploited, and thus DCF paths could be a powerful mechanism to focus analysis effort.

5.1. Hot Path Analysis

The preponderance of path analysis has historically been performed to identify "hot," or heavily executed, paths. This is common in compiler optimizations but is also used for testing purposes. The work of Vaswani et al. [30] defines a hardware-based programmable path profiling mechanism. This work focuses primarily on solutions for hot path analysis, limiting its adaptability to dynamic control frontier profiling. Buse and Weimer [5] utilize static analysis to identify hot paths which are determined to be over 50% of total runtime of an application and generated by only 5% of feasible paths. This work highlights the difficulty of path profiling before application deployment.

5.2. Path Analysis and Distributed Sampling

The concept of distributed sampling and end-users performing testing tasks has become a more prevalent topic. Greathouse et al. have demonstrated the feasibility of distributed sampling for otherwise heavyweight security vulnerability analyses. The applications are, however, dataflow analyses [13][14]. Ko et al. extensively investigate the concept of End User Software Engineering, which highlights the changing mindset of end-users playing a more involved role in the software life cycle [17].

Chilimbi et al. [7] have proposed a method to determine which paths were dynamically executed by deployed software that had never been tested, termed Efficient Path Profiling. This may be quite useful, but it focuses on finding latent bugs which are likely to directly impact users, thus focusing on software reliability. This is in contrast to DCF profiling, which seeks to enhance software security. A key assertion in this work was that edge profiling is sorely inadequate in comparison to path profiling. This built upon the previous work of Chilimbi et al. for Residual Path Profiling [8] which also focused solely on highly executed paths. Path-based data has been proposed by Liblit et al. to generate useful information on program crashes, specifically paths defined by conditional branches [20]. While this lends credibility to the usefulness of conditional branch-based path information, the purpose is strictly limited to post-mortem analysis of application failures. Ayers et al. [1] employ a different methodology to achieve these same ends.

5.3. Complementary Works

Testing technology has evolved along with software engineering techniques. Many useful tools exist which identify an ever-increasing ratio of bugs before deployment.

Godefroid et al. have implemented *DART* [10], a tool to automatically generate random tests to explore all possible code. This is a highly useful tool that could likely be made more effective with DCF profiling. Though it seeks to explore all sections of code, it cannot test all potential paths. A key challenge is that *DART* may never complete execution, making the determination of when to cease testing difficult.

The practice of fuzz testing supplies a software unit under test with a random generation of inputs in an attempt to "break" the unit, in the form of failed assertions and core dumps. The technique is sometimes called "black-box" testing because it creates inputs without regard to the internal structure of the software under test. This approach is very good in theory; however, in practice the probability of generating the correct set of inputs to achieve all possible paths within a given unit under test is effectively zero for non-trivial codes. Despite limitations, the approach has been effective at exposing security flaws. For example, Google's *cross_fuzz* tool generates random web pages for testing browsers, and it has exposed hundreds of potential security flaws in all major browsers [29]. When coupled with

dynamic program analysis tools that can identify security vulnerabilities without active exploits, such as taint analysis [28] or input bounds checking [19], fuzz testing becomes a power tool in the war against attackers.

While effective, pure random fuzz testing has limited penetration on complex program control sequences. Another important work related to DCF profiling is Microsoft's white-box fuzz testing tool *SAGE* [12]. This tool developed by Godefroid et al. strongly advances white-box fuzz testing of enterprise-level software. *SAGE* has become a primary tool for bug detection within Microsoft. The tool takes a test suite, with hand-generated and fuzz-generated tests, and then uses SAT-based techniques to derive new program inputs to change the direction of one branch in an existing dynamic code path. The newly derived code path is then subjected to symbolic execution analysis that includes input bounds checking, taint analysis and overflow checking. Approximately one-third of all Windows 7 security bugs found have been identified by *SAGE*. A highly representative example is a bug identified by *SAGE* which affected code that parsed ANI-format animated cursors [9]. The bug had escaped detection by extensive black-box testing over many years and generations of the Windows operating system. Using modest desktop hardware, *SAGE* was able to detect the bug within a few hours.

Random fuzz testing comprised the basis for testing four out of seven of our benchmark applications. Even so, we find vulnerabilities sensitized by DCF paths for these fuzz tested executions. The reality is that random fuzz testing does not provide deep code penetration [4][6][12]. This work is just another demonstration of the limitation of random fuzzing.

Even in light of such strong performance, many bugs are left undetected. A key challenge to any testing platform is the path space associated with a software application. Testing every path which may be executed remains infeasible for the foreseeable future. The infeasibility of complete path analysis is what makes DCF path analysis useful. Our work is to distill path data which may direct existing testing technologies. Applications such as *DART* and *SAGE* suffer the inadequacy of limited path exploration. The implementation of DCF path analysis can assist by directing such tools to high-value paths that likely contain security vulnerabilities.

Concolic execution tools allow deeper penetration of application code. However, these tools (such as *KLEE* [6]) have no path preference, including DCF paths. Indeed in achieving code coverage, *KLEE* will execute the basic block where the defect lies, but not necessarily with the path required to sensitize the bug. We fully expect this to be the case, as industry has currently moved into an era of full code coverage for test. This property of concolic execution, however, does not preclude discovering DCF paths anyway. Table 1 in Section 4.3 shows Schnauzer identified 17,351 length-4 DCF paths for *SQLite*, one of which sensitized the buffer overflow vulnerability identified in Table 2. This significantly narrows the field of discovery from the 13.6 million paths facing *KLEE*. As path lengths increase, the path space increases dramatically. The same measure for *SQLite* estimated almost 200 billion length-16 paths.

This further highlights how contemporary test can benefit from DCF analysis. Even when code coverage is achieved, vulnerability-enabling defects still remain. Current white-box testing attempts to brute-force application code to provide deeper penetration. DCFs provide a heuristic to narrow the path space faced by code penetration testing.

6. Conclusions

Bugs in software remain the greatest security threat in programs today. There is much compelling evidence in the testing literature (e.g., analysis of *Windows 7* security bugs[12]) which suggest that the key to finding and fixing security vulnerabilities is to analyze code paths at the dynamic control frontier. In this work we presented a comprehensive technique for profiling an application to discover the dynamic control frontier. We have shown that by using a distributed profiling approach, such profiling can be achieved efficiently for a substantial population of users. Furthermore, we have demonstrated the high value of DCF paths by correlating our discovered paths to 14 known security advisory vulnerabilities documented in the National Vulnerabilities Database. We feel strongly that efficient user-based dynamic control frontier path profiling, combined with existing white-box testing techniques and heavyweight dynamic security vulnerability analysis tools, will be a powerful weapon in the future fight against attackers.

6.1. Future Work

Opportunities exist to improve the profiling of the dynamic control frontier. Planned optimization of the current *DynamoRIO* and client implementation could yield further reductions in runtime overheads. A hardware-assisted profiling system is also planned to reduce to negligible levels the performance impact on end-user execution.

The next step in harnessing the dynamic control frontier is integration of profiled paths to existing test technologies such as *SAGE* or *KLEE*. This will be coupled with the deployment of optimized profiling for a long-running application interacting with a population of users. Together, this should work toward the discovery of yet-unknown security vulnerabilities.

Acknowledgements

The authors would like to thank the reviewers, whose insights improved this work. The authors acknowledge the support of the Gigascale Systems Research Center.

7. References

[1] A. Ayers et al., TraceBack: First fault diagnosis by reconstruction of distributed control flow, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 201-212.

[2] E. Bosman, A. Slowinska, and H. Bos, Minemu: The World's Fastest Taint Tracker, *Proc. of the Int'l Symp. on Recent Advances in Intrusion Detection (RAID)*, 2011.

[3] D. Bruning, Efficient, Transparent, and Comprehensive Runtime Code Manipulation, Massachusetts Institute of Technology, Ph.D. Thesis 2004.

[4] J. Burnim and K. Sen, Heuristics for Scalable Dynamic Test Generation, *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering},* Washington, D.C., 2008, pp. 443-446.

[5] R. P. L. Buse and W. R. Weimer, The road not taken: Estimating path execution frequency statically, *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 144-154.

[6] C. Cadar, D. Dunbar, and D. Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex System Programs, *Proc. of the Eighth Symp. on Opr. Systems Dsgn and Implementation (OSDI '08)*, 2008, pp. 209-224.

[7] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, HOLMES: Effective statistical debugging via efficient path profiling, *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 34-44.

[8] T. Chilimbi, A. Nori, and K.l Vaswani, Quantifying the effectiveness of testing via efficient residual path profiling, *The 6th Joint Meeting on European software eng. conf. and the ACM SIGSOFT symp. on the foundations of sftwre eng.: companion papers*, 2007, pp. 545-548.

[9] P. Godefroid et al., Automating software testing using program analysis, *IEEE Software*, vol. 25, pp. 30-37, 2008.

[10] P. Godefroid, N. Klarlund, and K. Sen, DART: directed automated random testing, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213-233.

[11] P. Godefroid and D. Lauchaup, Automatic partial loop summarization in dynamic test generation, *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*, 2011, pp. 23-33.

[12] P. Godefroid, M.Y. Levin, and D. Molnar, Automated Whitebox Fuzz Testing, *Proc. 15th Ann. Network and Distributed System Security Symp. (NDSS 08), Internet Society (ISOC)*, 2008.

[13] J.L. Greathouse, C. LeBlanc, T. Austin, and V. Bertacco, Highly scalable distributed dataflow analysis, *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, May 2011, pp. 277-288.

[14] J.L. Greathouse et al., Testudo: Heavyweight security analysis via statistical sampling, *41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 117-128.

[15] C. Huang and M.R. Lyu, Optimal testing resource allocation, and sensitivity analysis in software development, *IEEE Trans. on Reliability*, vol. 54, no. 4, pp. 592-603, Dec. 2005.

[16] B. Kitchenham and S. Linkman, Validation, Verification, and Testing: Diversity Rules, *IEEE Software*, vol. 15, no. 4, pp. 46-49, July 1998.

[17] A. Ko et al., The state of the art in end-user software engineering, *ACM Comput. Surv.*, vol. 43, no. 3, pp. 21:1-21:44, April 2011.

[18] D. S. Kushwaha and A. K. Misra, Software test effort estimation, *SIGSOFT Softw. Eng. Notes*, vol. 33, no. 3, pp. 6:1-6:5, May 2008.

[19] E. Larson and T. Austin, High coverage detection of input-related security faults, *Proceedings of the 12th USENIX Security Symposium (SECURITY'03)*, August 2003.

[20] B. Liblit and A. Aiken, Building a Better Backtrace: Techniques for Postmortem Program Analysis, University of California, Berkeley, TechReport CSD-02-1203, Oct. 2002.

[21] C. Luk et al., Pin: Building customized program analysis tools with dynamic instrumentation, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2005, pp. 190-200.

[22] T. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, 1976, pp. 308-320.

[23] Mitre. (2012, September) Common Vulnerabilities and Exposures. [Online]. http://cve.mitre.org/

[24] NIST, National Institute of Standards and Technology. (2012, September) National Vulnerability Database. [Online]. http://nvd.nist.gov/home.cfm

[25] H, Ohtera and S. Yamada, Optimal allocation and control problems for software-testing resources, *IEEE Transactions on Reliability*, vol. 39, no. 2, pp. 171-176, June 1990.

[26] A. Ozment, Improving vulnerability discovery models, *Proceedings of the 2007 ACM workshop on Quality of protection*, New York, 2007, pp. 6-11.

[27] R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed.: McGraw-Hill, 2009.

[28] E. Schwartz, T. Avgerinos, and D. Brumley, All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask), *2010 IEEE Symp. on Sec. and Prvcy*, May 2010, pp. 317-331.

[29] L. Seltzer, Microsoft, Google Clash Over IE 0-Day Leaked to Chinese Hackers, PC Magazine Online, 2011.

[30] K. Vaswani, M. Thazhuthaveetil, and Y.N. Srikant, A Programmable Hardware Path Profiler, *Proceedings of the international symposium on Code generation and optimization*, 2005, pp. 217-228.

[31] F. Wagle, Pu Calton, J. Walpole, and C. Cowan, Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, *DARPA Information Survivability Conference and Exposition (DISCEX)*, 2000.

[32] Z. Wang, K. Tang, and X. Yao, Multi-Objective Approaches to Optimal Testing Resource Allocation in Modular Software Systems, *IEEE Transactions on Reliability*, vol. 59, no. 3, pp. 563-575, Sept. 2010.

[33] Q. Zhao, D. Bruening, and S. Amarasinghe, Umbra: Efficient andn Scalable Memory Shadowing, *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, Toronot, 2010, pp. 22-31.

[34] H. Zhu, P. Hall, and J. May, Software unit test coverage and adequacy, *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366-427, Dec. 1997.

[35] T. Zimmermann and S. Neuhaus, Security Trend Analysis sith CVE Topic Models, *International Symposium on Software Reliability Engineering (ISSRE)*, 2010, pp. 111-120.