Research Statement

Stefan Bucur

January 2015

Software complexity has been continuously increasing. Systems expanded vertically, with new platforms and frameworks, and horizontally, with an abundance of readily-available libraries. Many applications today run on a cloud infrastructure, use a web framework, and have a browser or mobile client, all while putting together code from several sources. Ensuring that this software amalgam behaves as intended is crucial, as we are increasingly entrusting it with our personal data and daily operations. Alas, bugs are prevalent and software vulnerabilities are making headlines more often than ever.

My research is about developing system techniques that scale automated program analysis, such as bug finding and test generation, to real-world software stacks. I am interested in pragmatic approaches that work on systems as they are written today. At the same time, I am targeting the long-term goal of making future system development more amenable to program analysis.

To date, I have focused on symbolic execution, a thorough technique attractive for its soundness (no false positives) and flexibility in focusing on the relevant execution paths. In a nutshell, symbolic execution works by replacing the concrete inputs of a program with symbolic ones, representing the set of values allowed by their type. When a branch involves symbolic values, the program state "forks" into clones that proceed independently. Recursively, this creates an *execution tree*, whose paths are prioritized by a *search strategy* to meet a goal, such as statement coverage or bug finding. There are two major hurdles preventing the scalability of symbolic execution to large systems: (a) *path explosion*, i.e., the number of execution tree paths being roughly exponential in the number of program branches, and (b) *the environment problem*, caused by the program invoking code outside the scope of the analysis (e.g., kernel system calls).

In my research, I am addressing the two scalability problems using system techniques aimed at narrowing and controlling the program execution space. I designed symbolic execution interfaces for the POSIX operating system environment, as well as for interpreters executing high-level dynamic languages. At the same time, I developed general techniques for alleviating path explosion, such as a search strategy technique that confines the program "hot spots" causing path explosion, and a symbolic execution parallelization algorithm that is the first to linearly scale on clusters of commodity hardware. I embodied these techniques in tools that are open-source and have grown a community of researchers and practitioners.

Symbolic Execution for Software Stacks

My work has focused on two areas of the software stack: low-level systems that make rich use of the operating system interface and programs written in interpreted languages, such as Python or JavaScript.

Cloud9 [3, 5] is a symbolic execution platform that provides efficient support for the operating system (POSIX) interface used by low-level systems, such as web servers, multi-process systems, or language runtimes. Executing these systems symbolically is challenging. When handling system calls, an engine has two straightforward choices: either concretize symbolic data and pass it to the external operating system, hence losing completeness, or bundle the operating system code with the target program, which exacerbates the path explosion. Instead, Cloud9 builds on the insight that, for the purpose of automated testing, a large

fraction of the interface can be compactly modeled as guest code on top of *three basic abstractions*: threads and processes, synchronization, and address spaces with shared memory. These abstractions are built into the symbolic execution engine and provided to the guest code as "symbolic system calls." They are used as building blocks for implementing much of the POSIX interface as a model library replacing parts of the standard C library.

On the developer side, Cloud9 is controlled using *symbolic tests* that define the scope of the analysis. A symbolic test is a fixture and test logic—akin to a unit test—that runs inside the symbolic environment and uses POSIX-specific symbolic abstractions. For example, a symbolic test can configure the behavior of the symbolic thread scheduler to simulate different concurrency conditions, the read size fragmentation for a file descriptor, or symbolic fault injection (i.e., exploring both a successful and failed POSIX call through state branching).

The symbolic system calls, the compact POSIX model, and symbolic tests enabled Cloud9 to find security vulnerabilities and other bugs in complex system software, such as memcached[1], lighttpd, and curl. Cloud9 is available at http://cloud9.epfl.ch and has grown a community of over 50 researchers and practitioners on its discussion group.

Chef [2] is a platform for obtaining symbolic execution engines for interpreted languages. Manually writing such an engine is laborious and unsustainable, as language semantics are typically complex, lacking precise documentation, and continuously evolving. Moreover, some language runtimes deliberately depart from specifications, such as JavaScript browser VMs, in a quest to stay competitive on the market. To work around these issues, Chef provides a *platform* for symbolic execution of interpreted languages where the semantics of a language are provided by plugging its interpreter. In turn, the platform acts as a symbolic execution engine that *runs correctly by construction on all programs written in the language*.

Chef uses the interpreter as an "executable specification," by running it inside a low-level symbolic execution engine (e.g., x86 or LLVM). However, running the interpreter as a regular program is ineffective, as the exploration "gets lost" in the implementation details. To accommodate this, Chef structures the execution space of the interpreter by collecting the address of each program instruction executed, i.e., its *high-level program counter* (HLPC). For each interpreter path, the sequence of HLPCs defines the high-level path through the interpreted program. This information is leveraged by the search strategy of the low-level symbolic execution engine, to maximize the number of high-level paths discovered.

With Chef, we obtained engines for Python and Lua that were effective at generating high-coverage test suites and discovering bugs in popular library packages; we even executed Python PaaS web applications [1]. Chef and the Python and Lua engines are available at http://dslab.epfl.ch/proj/chef.

Tackling Path Explosion

In my work on Cloud9 and Chef, I developed several general techniques for alleviating path explosion.

Class-Uniform Path Analysis. We empirically noticed that path explosion manifests as a set of "hot spots" in the code, where states fork significantly more often than the rest of the code—for instance, loop conditions depending on symbolic input. This biases the state population and pushes the execution into a vicious circle: the more hot-spot states in the queue, the more likely for a strategy to select them, which further amplifies their growth. Based on this observation, class-uniform path analysis (CUPA) is a strategy framework whose basic idea is to group execution states into classes, then have the selection algorithm first run uniformly on the set of classes, then on the states in the selected class. The rationale is that any source of path explosion that biases the set of states is now contained in the CUPA class it originates from, thus allowing the states in the other classes to make progress in the exploration. The choice of the class partitioning depends on the

[1] CVE-2011-4971

global exploration goal. For instance, Chef employs a two-level CUPA strategy, where the first class is the state high-level program counter, and the second is the low-level x86 program counter. As a result, CUPA helped discover up to $1000\times$ more high-level paths than using a default depth-first state selection strategy.

Parallel Symbolic Execution. Another way to tackle path explosion is to "throw hardware at the problem" through massive parallelization. I developed a parallel symbolic execution algorithm that is the first to linearly scale on clusters of commodity hardware, such as clouds and modern data centers. This benefits both complex individual runs, as well as deployments as an automated testing service [4]. Parallelization can also be employed in conjunction with other techniques, such as state merging [6]. Parallelizing symbolic execution is challenging, as the shape of the execution tree is not known a priori. To avoid worker machines becoming idle, a load balancer maintains an aggregate view of the global exploration progress, obtained from periodic worker updates. Each worker maintains a view of the execution tree, where node are labeled either as local candidate for execution, or blocked as being executed remotely. Load balancing involves blocking some of the source nodes, serializing them, and reconstructing them as candidates at destination by replaying their path in the execution tree. Blocked nodes are not destroyed, in order to make replay efficient if states are transferred back to the worker.

Future Work

I want to continue developing system techniques that make program analysis useful for the software running today. At the same time, I want to make future systems more amenable to program analysis.

Lifting Semantics from Implementations. Conceptually, Chef uses the language implementation as implicit semantics for symbolic execution. I want to take this general idea further and automatically *lift (extract) explicit semantics* from an implementation. As systems become larger and span more layers of abstraction, their functional and security properties are also more often expressed at a high level. However, it is the low-level representation that gets executed on the machine. A semantics lifting tool takes as input an implementation and a "template" of the high-level semantics model and outputs a concrete instance of the template. This output can be compared to an existing specification, or used as-is, as autogenerated documentation, or passed to an automated high-level bug finding tool.

For example, a *grammar lifting* tool reconstructs a formal grammar from a parser implementation. Parsers are sensitive for system security, as they often are the first gate for unsanitized input before it advances in the system. A reconstructed grammar could have helped reveal the Shellshock bug in Bash at integration testing.

Another example is *string lifting* for interpreter analysis. Strings are first-class values in high-level language, but the actual execution is performed on memory bytes by the interpreter and its environment. String lifting maps arbitrary low-level string operations to higher level string primitives (e.g., string concatenation or regular expression matching) that can be analyzed more efficiently. This can help in security analysis. For instance, many critical XSS vulnerabilities are caused by bugs in the browser implementation, which is invisible at the JavaScript level, but can be exposed by string lifting.

Abstract Interpretation from Concrete Semantics. Static analysis, and type inference in particular, is helpful for finding programming errors in dynamic languages, as they commonly lack the type information available to compiled languages. However, static analysis engines are implemented by hand, hence sharing the challenges of building a symbolic execution engine. To this end, I want to take the "interpreter as executable specification" approach in the realm of abstract interpretation. Instead of writing the transition functions by hand, developers only describe the abstract domain as conversion functions between a symbolic state and an abstract state. The abstract engine performs the transitions by executing the interpreter using a symbolic state and invoking the transformation functions.

Thinner Abstraction Layers. One way to make systems more amenable to program analysis is to split abstraction layers into multiple smaller layers that can be analyzed in isolation. For example, a possible splitting point is program memory management. Most systems today do not explicitly manage their virtual address space, but divide it into logical chunks—*memory objects*—and resort to memory allocators to encapsulate the memory object management. However, the semantics are still complex, as program pointers still refer to the underlying address space abstraction. Similar to how virtual memory brought convenience and isolation between processes, I want to explore the design space of binary-level virtualized memory objects. For instance, instead of holding address space pointers, C pointer types store opaque tokens and offsets into memory objects. Only when needed, the operating system pins the objects in memory and transparently converts tokens to regular pointers, similar to how virtual memory addresses are paged in and translated to physical addresses. This mechanism opens more opportunities than just providing memory protection. First, program analysis is more convenient, due to simplified alias analysis and explicit memory objects in the data flow. Second, the object storage also becomes flexible; for instance, sensitive memory objects could be stored encrypted in memory when they are not used, and be decrypted only when pinned in memory.

The ultimate goal of my work is to make formal techniques broadly adopted in practice. My approach has been to start with a concrete dependability problem, identify a possibly effective formal approach, then design a solution that accommodates the idiosyncrasies of the targeted systems. My work demonstrated this is achievable for symbolic execution by using system techniques that blend the reduction of path explosion with user-facing flexibility. I am looking forward to continuing my work in other areas of system dependability.

References

[1] Stefan Bucur, Johannes Kinder, and George Candea. Making automated testing of cloud applications an integral component of PaaS. In *Asia-Pacific Workshop on Systems (APSYS)*, July 2013.

[2] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.

[3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, April 2011.

[4] George Candea, Stefan Bucur, and Cristian Zamfir. Automated software testing as a service. In *ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[5] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. In *ACM Operating Systems Review*, volume 43, December 2009. Also in Proceedings of the 3rd SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS), Big Sky, MT, October 2009.

[6] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2012.