Trust, but Control: A Paradigm Shift in Program Analysis

Andreas Zeller · Konrad Jamrozik · Philipp von Styp-Rekowsky
Center for IT-Security, Privacy and Accountability (CISPA), Saarbrücken, Germany
{zeller,jamrozik,styp-rekowsky}@cs.uni-saarland.de

ABSTRACT

In the next decade, program analysis will be facing substantial and unsurmountable issues. *Analysis of third-party programs* will be hard to impossible, as programmers can deliberately make their programs unanalyzable, posing major risks for security assessment. The issues are due to *fundamental limitations* of code analysis (e.g., the halting problem) and execution analysis (i.e., incompleteness), which cannot be overcome by better methods. In real-world settings, this effectively disables our ability to predict program properties. On top, all techniques need detailed formal *specifications* of expected and unexpected properties, which hardly exist.

To overcome these problems, we suggest a strong research focus in the following areas, which so far are not part of the program analysis portfolio: *Specification mining* to infer what a program "normally" does; *Test generation* to systematically explore behavior for specification mining; and *Sandboxing* to prohibit abnormal behavior at runtime. In early experiments, the combination of the three has shown promising results: *Sandbox mining* constrains a program to *the behavior found during testing*, thwarting common malware, attack, and backdoor strategies.

1. INTRODUCTION

Ensuring that a program conforms to its specification always has been a fundamental challenge of computer science. Consequently, a wide array of techniques has been developed to verify software correctness. Most notably, *static code analysis* symbolically verifies program code against desirable or undesirable properties, while *software testing* checks *executions* for desired or undesired results. Both techniques are widely used in practice, and scale well enough to provide sufficient confidence into today's software systems and components. Important advances of the last decade include symbolic analysis techniques that allow for semi-automatic or even fully automatic proofs of program properties, as well as automatic test generators that can cover thousands of behaviors in seconds.

All these techniques, however, rely on one assumption—namely that it is the *creator of the software* that runs them. And it is only under this assumption and its consequences that analysis and testing can work at all. Static analysis assumes that code is accessible and analyzable in the first place; whereas testing assumes that one execution is representative for several similar executions. Neither is a problem if we own the source code in the first place, and if we are interested in getting it analyzed and tested.

One of the greatest successes of Software Engineering is that modern software design principles make code *interoperable* and *reusable* to an extent never known before. The same mechanism that drives our productivity, though, is the same mechanism that imposes risks. If we do *not* own the code, reverse engineering is difficult (e.g, of binary code), if not impossible (e.g., of remote code). What we can do is *test* the third-party program; but then, incompleteness always leaves us with the risk of unexpected behavior—be it by accident or on purpose.

In the end, we can only make use of third party code if we can *trust* its provider. The lack of control, however, becomes a problem if our trust is fooled. If the code is *adverse,* it is in the interest of its provider *not* to support analysis or testing. This is when the limitations of code analysis and testing turn into *weapons* that can be easily turned against us: *Program creators have every means at their disposal to make program behavior deliberately unpredictable.* Code analysis, for instance, can be thwarted by including practically unpredictable code such as an interpreter. Testing can be fooled by including behavior that is triggered only under specific circumstances—such as a specific location or date. These limitations are fundamental: *Whatever improvements we have in analysis and testing techniques, there always will be a way to thwart them.*

What we propose in this paper is a *paradigm shift for program analysis:* Rather than trying to *predict* program behavior before production, we should focus on *preventing* program behavior at runtime. Specifically, we suggest a research focus in the following areas, which so far are not part of the program analysis portfolio:

Specification mining to infer what a program "normally" does. This relieves us from the problem of manually specifiying what is allowed and what is not.

Automatic test generation to systematically explore program behavior—not so much for finding bugs, but rather as an input for specification mining.

Sandboxing to prohibit undesired behavior at runtime; most notably *behavior not seen during testing.*

The combination of these three could effectively prevent *unexpected behavior changes* such as those caused by latent malware, vulnerability exploitations, malware infections, or targeted attacks. It would also protect against *backdoors* that would not be discovered during normal usage. The only way for malware to avoid being blocked by the sandbox is to *fully activate during testing already*, dramatically increasing the chances for analysis and detection. Mined sandboxes would require no changes to existing programs or processes—and thus have a chance of actually being used in practice. Finally, on a conceptual level, we obtain *guarantees from testing*—namely the absence of any behavior not seen during testing.

To get there, though, we will still need major advances in these three areas, supported by strong and scalable code analysis and testing techniques. As a first experiment, we have implemented the combination of the three in a tool called BOXMATE for Android apps that demonstrates the feasibility of the approach.

2. THE CHALLENGE

What is it that makes program analysis of third-party code so hard? To answer this question, let us reconsider the fundamental limitations of analysis and testing.

2.1 The Halting Problem

The halting problem implies that there cannot be a universal machine that predicts the behavior of any given program in finite time. In pratical code analysis, this leads to *overapproximation,* resulting in more behaviors being flagged as possible than could actually take place. A sound static analysis can prove several behaviors to be impossible, though, and reduce the space of possible program behaviors, even in presence of overapproximation. This is how static analysis works in practice, and successfully so.

Static analysis reaches its limits as soon as third-party code is involved. Such code may come in binary form, which calls for a specialized analysis. It may reside on a remote server, which trivially makes code analysis impossible. Or it may include an interpreter, which brings in the halting problem in its full glory. All of these make code analysis hard to impossible; but we may still ask the third party to run similar analyses and rely on the provided guarantees.

All of this breaks down in an *adverse* setting, where all limitations of code analysis are turned against us. Mechanisms that prevent static analysis are all too common in recent malware. As an example, consider the ROMBERTIK malware [1], hooking into the browser to propagate keystrokes to an attacker controlled server. Disguised as a mail attachment, ROMBERTIK first decrypts itself when opened, then launches a second copy of itself, overwriting the original copy with the actual malware. The resulting executable contains over 8,000 functions, all with non-standard control flow, obfuscating the code actually executed. It takes researchers days of manual work to understand how a program like ROMBERTIK works; it would be easy for malware writers to arbitrarily extend this.

Should we simply prevent using software we cannot understand? Possibly. But keep in mind that every nontrivial software is the result of years and years of research and development, all encoded into the program. Disclosing its secrets is a huge economical risk for every vendor. It is in the vendor's interest to keep things hidden—and unanalyzable. Unfortunately, the same interest is also shared by malware writers. *An adverse program can always avoid code analysis, an interest shared by commercial software.*

2.2 The Incompleteness Problem

If the halting problem is the bane of code analysis, incompleteness is the fundamental limitation of testing. As much as we test, we can only analyze a finite sample of executions in an infinite space of possibilities; and there is always the risk that while all our tests have passed, the next execution fails.

The reason why testing works in practice is that we can usually assume that the software will behave similarly for wide ranges of equivalent inputs, and therefore, a single representative will "cover" this entire range. This assumption holds because this is how software is typically being written, and as long as the test in question covers sufficiently many behaviors, we can gain a sufficient confidence in our testing results.

All this confidence gets lost, though, if we cannot trust the writers of the software. It is easy for developers to detect that software is being tested; and if so, to block behavior that would make the test fail. In the 2015 Volkswagen scandal, the car engine software would detect that the engine was being tested, and consequently reduce emissions to conform to the test. Outside of the testing environment, the engine would behave differently, yielding more power, but also up to a 40-fold increase in emissions.

Malware writers can easily exploit incompleteness by activating malicious behavior only after some time, in a specific environment, or when not being analyzed, each of which thwarts observation during testing. If the ROMBERTIK malware detects a process whose name contains "malwar", "sampl", "viru", or "sandb", it aborts. Otherwise, it writes random data to memory for 30 seconds; if the test is not run long enough, it will not capture the malicious behavior that activates later—including wiping the hard disk if ROMBERTIK finds its binary has been altered. No matter how sophisticated testing may be, *an adverse program can always avoid detection.*

2.3 The Specification Problem

Even if we were able to thoroughly analyze and test third-party code, we would still face a major problem: Which are the properties we are supposed to check? As of now, most analysis techniques have focused on generic properties that apply to all programs. An integer variable must not hold a string. A null pointer must not be dereferenced. Array bounds must be respected. Heap memory must be allocated before use. Any of these properties can be arbitrarily hard to check and prove; but all of them are easy to express.

If we want to check some application-level property, though, we first must *specify* this very property—in a formal description, because only then would we be able to pass the specification to an analysis or testing tool. We know pretty well how to specify (and consequently prove) individual functions by means of conditions and invariants, usage of such specifications in actual applications is still slim, as indicated by the low usage frequency of `assert()` statements in open source code, or the almost nonexisting integration of contracts in common programming languages.

The problem becomes worse as we want to specify *application-level* or even *system-level* properties. As an example, consider *information flow:* Which information is allowed to flow from one component to another? How would its usage in the respective components be restrained? Who in which role under which condition can access this information? All of this *can* be specified, and all of this *can* be tested and analyzed against; and we may be able to detect that the ROMBERTIK malware violates some specification by having keystrokes flow to an unauthorized third party. However, note that there are several legitimate usages for sending keystrokes over the network, so a specification therefore must separate the benign from the malicious. And what about all the other sensitive data? Where is it allowed to go to? Who specifies all that?

If we start with a clear specification, and derive an application from it, proving its conformance along the way, we may get software that is correct by construction. But if an application has *not* been developed with such specifications in the first place (as is likely the case for most applications), retrofitting such specifications will be a major problem, let alone evolving them along with the application. *The cost of formal specification is too often conveniently ignored.*

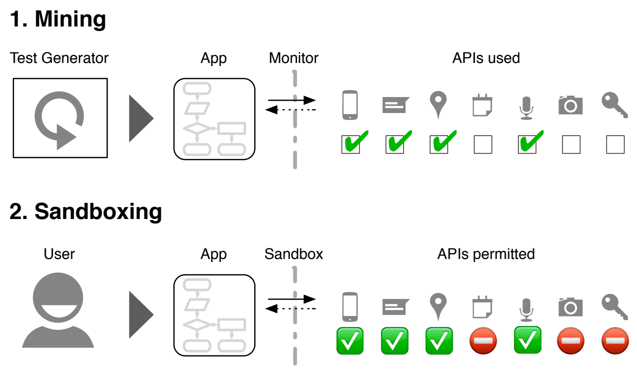

Figure 1: Sandbox mining in a nutshell. The mining phase automatically generates tests for an application, monitoring the accessed APIs and resources. These make up the *sandbox* for the app, which later prohibits access to resources not accessed during testing [2].

3. A SOLUTION: MINING SANDBOXES

As long as we can assume that code is analyzable, testable, and well-specifiable, everything is fine; and we as researchers can continue to refine our techniques in our lab experiments. In the real world, however, we will be increasingly faced with systems where there is no central control, and where individual system components all come with their own operating software—whose code may or may not be analyzable for outsiders.

The only assumption we can make is that we can *execute* a program and, consequently, *test* it; and while we are executing it, we can *track* its interaction with the outside world, for instance by monitoring API calls, or basic input/output. (In practice, there may be many more features to observe, such as internal state or data flow; but for now, we stick to the basics.) These findings can feed dynamic analysis, providing and summarizing insights into what happens in these executions. By construction, these insights are incomplete, and other (in particular malicious) behavior is still possible. The key idea is to *turn the incompleteness of dynamic analysis into a guarantee*—namely by having a sandbox *enforce that anything not seen yet will not happen.*

Let us illustrate this with a simple application. On the Android platform, for instance, developers have to declare that an app needs access to specific resources. The popular SNAPCHAT picture messaging application, for instance, requires access to the Internet, the camera, and the user's contacts; these permissions would be reviewed and acknowledged by the user upon download and install. If an application fails to declare a permission, the operating system denies access to the respective resource.

Our BOXMATE tool now works in two phases (Figure 1):

Mining. In the first phase, we *mine* the rules that will make the sandbox. We use an *automatic test generator* to systematically explore program behavior, monitoring all accesses to sensitive resources.

Sandboxing. In the second phase, we assume that *resources not accessed during testing should not be accessed in production either.* Consequently, if the app (unexpectedly) requires access to a new resource, the sandbox will prohibit access, or put the request on hold until the user explicitly allows it.

During systematic GUI testing, the mining phase determines that SNAPCHAT indeed requires access to the camera, location, Internet, and so on. The resulting sandbox can be *much more fine-grained* than the original Android sandbox, and easily prevents a number of otherwise permitted attack schemes. Compromising all contact data, sending text messages in the background, continuously monitoring the audio or the device location, would all be disallowed, simply because this behavior is not what we find during testing.

Even more important, though, is that the sandbox also protects the user against *unexpected behavior changes.* Assume an app like SNAPCHAT was malicious in the first place, and placed in an app store. Then, the attacker would face a dilemma. If the app accesses all contacts right after the start, this would be detected in the mining phase, and thus made explicit as a sandbox rule permitting behavior; such a rule ("This app reads all contact details in the background") could raise suspicions even with non-expert users, because there is no apparent functionality in SNAPCHAT that requires this. If, however, the app stayed benign during mining, it would be disallowed from accessing contact details in production, except for phone numbers during the "Find friends" functionality.

In other domains, the ROMBERTIK malware would face a similar dilemma: If it aborts execution, because it detects that it is being tested, the resulting sandbox would grant no access to any resource, rendering ROMBERTIK harmless for users, and useless for its creators. And if we extend the sandbox to *sensor values*, a sandbox could easily detect that the Volkswagen engine software behaves very differently in production and when being tested.

4. A RESEARCH AGENDA

The proposed combination of test generation and sandboxing addresses all the challenges listed in Section 2. As a dynamic technique, we do not suffer from the halting problem; specifications are inferred automatically; and the incompleteness of testing is turned into an advantage. All of this, however, depends on a number of assumptions which translate into three challenges:

Can test generators sufficiently cover behavior? Obviously, any behavior missed during mining will result in a sandbox warning as soon as this behavior is triggered during production. In our Android example, for instance, if we miss that SNAPCHAT can shoot videos, then the sandbox will ask for confirmation as soon as this functionality is asked for later.

We thus need test generators that very systematically and efficiently explore normal behavior to avoid false alarms. Note that in contrast to traditional testing, which focuses on finding bugs, the goal of covering normal behavior would be easier to achieve. *All these call for major improvements in systematic test generation.*

Can we sufficiently reduce the attack surface? The counterpart to false alarms is *missed alarms.* These can take place if the mined rules are too *coarse.* In our application, once we learn that SNAPCHAT accesses the user's contacts, such an access would also be possible from some background process within SNAPCHAT.

We can counter this issue by creating more fine-grained rules, for instance, by associating resource accesses with the GUI elements that trigger them. Thus, we would find and enforce that SNAPCHAT accesses contacts only when the user presses the "Find friends" GUI button; and it only accesses the friends' phone numbers. Such fine-grained rules reduce the attack surface—but also increase the effort for mining them all. To obtain 100% accuracy, the sandbox rules would have to be Turing complete, just as the program they apply to; and then, the sandbox would suffer from the halting problem as well. Therefore, *inferring accurate rules that describe* *program behavior will be a central challenge for specification mining.*

Can we check rules efficiently? As sandbox rules are checked during production, efficiency is a major issue. Of course, the whole point of predicting program properties is to avoid having to check them at runtime. However, it is this striving for maximum efficiency that has brought us the many security issues we see today; and therefore, the widespread absence of runtime checks may have to be thoroughly reassessed.

Controlling resource accesses, as implemented in BOXMATE, has no measurable impact on execution time, as we add one single check on top of a system operation that is expensive in the first place. If we were to assess other properties, such as dynamic data flow or data consistency, the impact would be more substantial. We therefore need to work on *strong static and symbolic analysis* that helps eliminating and reducing runtime checks, as well as *efficient sandboxing techniques*, possibly relying on hardware support for compartmentalization and range checking.

In a first set of experiments [2, 3], BOXMATE requires less than one hour to extract a sandbox from an Android app, with few to no confirmations required for frequently used functionality—and, of course, dramatically improved security by constraining what the app can do. On top, we found the resulting rule sets easy to comprehend, and to compare across app versions; this way, *even laymen may also identify and track changed behavior as it comes to sensitive resources.* All these results are promising; and we want to bring them to other domains, such as smart home components, embedded systems, car components, or servers.

5. FREQUENTLY ASKED QUESTIONS

I am a vendor. How can I ensure the mined sandbox encompasses all legit behavior? Modern test generators are well set to achieve high coverage; future test generators will get even higher coverage faster. You would top this with your own set of tests, and ship the mined sandbox with your program.

How will your approach ever get 100% coverage? Modern test generators are already very effective at covering "normal" behavior, which is just what we need. Sensitive resource accesses, our most important target, is a small subset of behavior that is relatively easy to cover.

Test generation may not cover exceptional behavior. "Benign" exceptional behavior would only rarely access yet uncovered sensitive resources—in sharp contrast to backdoors.

How can I protect my intellectual property? By design, the approach relies only on externally visible dynamic interaction; the implementation can remain unchanged—and obfuscated.

I want to write malware. How can I stay in business? With mined sandboxes, you are in a "disclose or die" dilemma. Either you expose malicious behavior during mining, and then it becomes explicit for scrutiny and discussion; or you do not, and then the sandbox prevents it.

How can you distinguish benign from malicious behavior? To be allowed by the sandbox, any malicious behavior must also be present during testing, and becomes explicit in the rule set; both allow for easy detection and assessment.

Can't I ship a permissive sandbox with my malware? Users of mined sandboxes can safely assess your program and its sandbox before installation. See "How can I trust a supplied sandbox?", below.

I could craft an "official" rule that allows my attack. Your rule would have to withstand public scrutiny, very much like open source programs and their changes.

I am a user. I am getting a false alarm. What can I do? Use an "official" sandbox provided by the trusted vendor. Or re-mine the sandbox to have it include the legitimate behavior.

How can I trust a supplied sandbox? You can mine a sandbox yourself and compare its rules against the supplied sandbox; if the supplied sandbox allows more behavior than your sandbox, there should be a legitimate reason.

If the test generator exercises a backdoor during mining, it becomes part of the sandbox, right? A backdoor is typically designed such that testing would not find it. If testing can find it, so can the user, and then it is more likely to be legit.

6. CONCLUSION AND CONSEQUENCES

As our applications more and more rely on third-party code, our current means to predict program behavior are increasingly challenged—not only *practically* because of scalability issues, but also *fundamentally,* as limitations such as the halting problem or the incompleteness of testing can thwart any behavior prediction. This problem is emerging today; in 2025, it will be in its full bloom.

Rather than attempting to improve our prediction capabilities, we should focus on our *enforcement* capabilities, ensuring that program behavior is confined to what can be expected. In this light, the combination of specification mining, test generation, and sandboxing is particularly promising. In our approach, we give testing a new purpose, namely to extract *normal* behavior—a task that testing arguably does much better, and even more so in the security domain. By excluding *behavior not seen during testing,* we turn the incompleteness of testing into a guarantee that bad things not seen so far cannot happen.

Still, it will take a decade until the prerequisites—test generation, specification mining, and efficient sandboxing—can evolve from research prototypes into widely applicable tools and frameworks in several domains, leveraging the extensive expertise that exists in code analysis and software testing. We sincerely hope that this shift will come in time for the hyper-scale, hyper-opaque, and hyper-distributed systems we will have by then—because otherwise, we would have to admit that our programs are out of control.

For more information on mining sandboxes, including technical reports and all experimental data, see our site

http://www.boxmate.org/

Acknowledgments. This work was funded by an European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing". Sudipta Chattopadhyay, Andreas Rau, and Kevin Streit provided helpful comments on earlier revisions.

7. REFERENCES

[1] BAKER, B., AND CHIU, A. Threat spotlight: Rombertik—Gazing past the smoke, mirrors, and trapdoors. *Cisco Blogs* (May 2015). http://blogs.cisco.com/security/talos/rombertik/.

[2] JAMROZIK, K., VON STYP-REKOWSKY, P., AND ZELLER, A. Mining sandboxes. Tech. rep., Saarland University, 2015. Submitted to ICSE 2016, technical track.

[3] ZELLER, A. Test complement exclusion: Guarantees from dynamic analysis. In *Proc. International Conference on Program Comprehension (ICPC)* (2015). Abstract of invited keynote.