How Exhaustive Static Analysis can Prove the Security of TEEs

TRUST IN SOFT

Executive Summary

A **Trusted Execution Environment**, or TEE, is a key component in many of the devices we own and use.

The **purpose** of a TEE is to guarantee the authenticity and confidentiality of its code, data and runtime states, and thus ensure the reliability and security of the device in which it resides. It is an isolated environment within a processor where, even if the operating system is compromised, your data is protected.

To **fulfill its purpose**, a TEE must be (a) perfectly reliable in the execution of its functions, and (b) impervious to unauthorized access by any means, while still being able to accept and use data from external sources.

But a TEE is made up of code—code developed by humans. And since humans are prone to make mistakes—especially when engaged in complex tasks like computer programming—code is prone to have defects (bugs). Bugs make code unreliable. They present opportunities for hackers to exploit. Thus, for a TEE to reliably fulfill its function, all bugs within its code must be eliminated.

Unfortunately, due to ever-increasing size and complexity of today's software applications, removing bugs has become increasingly difficult. In fact, it has become virtually impossible to eliminate all bugs from a typical application using traditional software testing methods. The level of effort required is simply too high to be cost-effective.

Luckily, a new technology has emerged that can reliably detect and help eliminate all coding flaws within a software application, and also guarantee the application complies perfectly with its specification. It is a technology ideally suited to **validating a trusted execution environment.**

Exhaustive static testing is a methodology and a framework that applies a variety of **mathematical formal methods** to the task of software validation. It can help **guarantee the absence** of common **coding flaws that can be exploited by hackers** and cause software to behave in an unpredictable manner. Plus, it does this in a way that is completely **transparent** to the user and in no way disrupts the normal software development process.

What's more, exhaustive static testing can also be used to **guarantee that your trusted execution environment** or other critical code complies perfectly with its specification.

This white paper will explore:

- The challenges of guaranteeing the reliability, integrity and security of a TEE
- Why traditional software testing is inadequate for this task
- How those challenges can be overcome with exhaustive static analysis
- It will also look briefly at some examples of how exhaustive static analysis is currently being used in industry to develop trusted execution environments, as well as some key attributes to look for in an exhaustive static analysis solution.

The need for trust... and a trusted execution environment

Twenty-first century technology is increasingly complex, software-driven, and connected. Plus, consumer technology is becoming increasingly personalized, carrying more and more sensitive data that must be protected.

We need to be able to **trust our devices** to protect both themselves and our personal data from malevolent hackers. In essence, we need our devices to guarantee a high level of trust.

A trusted execution environment (TEE) is a secure area within a processor designed to provide the level of trust we require. It is an environment in which the code executed and the data accessed are both isolated and protected. It ensures both confidentiality (no unauthorized parties can access the data) and integrity (nothing can change the code and its behavior).[1] Figure 1 illustrates the partitioning of a trusted execution environment ("Secure World") within a processor.

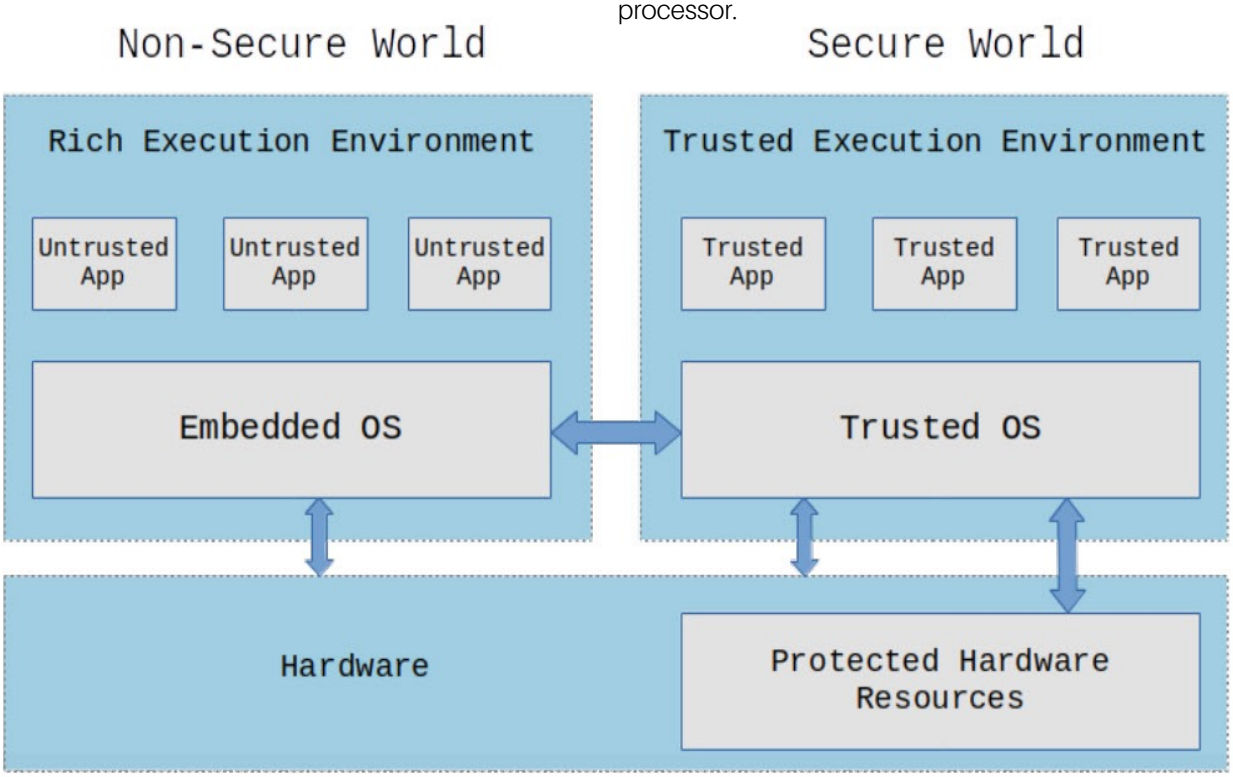


Figure 1: Partitioning a system using a trusted execution environment (Source: #embeddedbits)[2]

TEEs are essential components of many of the devices we own and use, including smartphones, tablets, game consoles, set-top boxes and smart TVs. They are well-suited to providing security for applications like storage and management of device encryption keys, biometric authentication, mobile e-commerce applications, and digital copyright protection.

For a TEE to fulfill its function, however, the behavior of its code must be perfectly deterministic, reliable, and impervious to attack. Therefore, it must be free of software errors that could be sources of anomalous behavior and vulnerabilities for hackers to exploit.

The challenges of validating a trusted execution environment

TEEs are made up mostly of code. Lots and lots of code. While not as extensive as rich operating systems, they're still complex beasts.

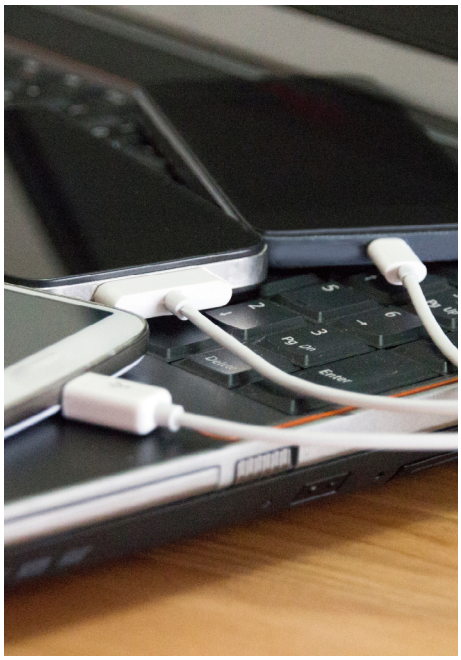
When we write code, we introduce **bugs.** That's simply inevitable, due to the complexity of today's applications and the propensity for human error in complex processes like software development. The larger and more complex the piece of code, the greater the number of bugs introduced, and the more difficult it is to detect and eliminate those bugs.

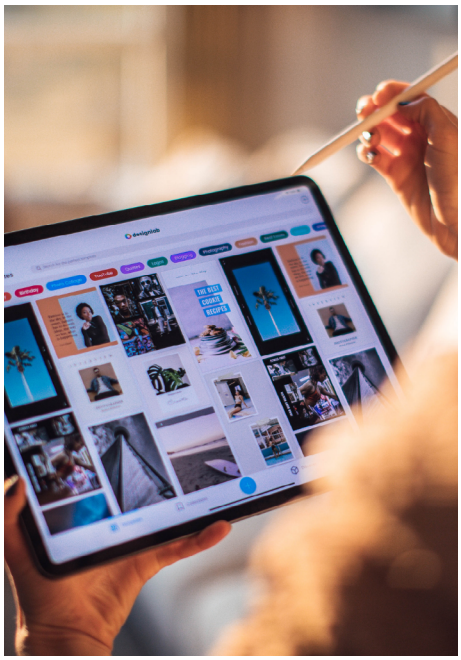
According to data pipeline management and analytics firm Coralogix: [3]

- Developers, on average, create 70 bugs per 1000 lines of code
- 15 bugs per 1,000 lines of code find their way to customers
- Fixing a bug takes 30 times longer than writing a line of code
- 75% of a developer's time (1500 hours/year) is spent on debugging

For many applications—most consumer apps, for example—removing bugs is not of great concern. As long as a bug isn't a showstopper, it can be cleaned up in a future release.

For applications where safety, reliability or security are critical, however, failure to find and remove bugs could lead to disaster. A TEE is one such application. The problem, then, is finding an efficient way to eliminate those undefined behaviors.





The drawbacks of traditional software testing

Traditional software testing consists of taking the software requirements, which have been developed by decomposing and refining the top-level system requirements, developing a test plan and test cases which cover those requirements as thoroughly as possible, and then testing the software against those requirements. You take the requirements and brainstorm your test cases. At the end, you hope that the tests you've performed have adequately verified that the software meets its requirements and that the code exhibits no **anomalous behaviors** under real-world conditions.

The process is an iterative one based on finding bugs and fixing them. It might also be called a "best efforts" process; you do the best you can to find bugs that matter the most in the time you have budgeted for software testing.

This traditional process, however, offers no guarantees that all bugs have been eliminated.

The common tools on the market supporting traditional software testing are typically designed to uncover obvious errors. Most search for patterns recognized as bad coding technique. Unfortunately, such tools are not designed to find more uncommon, subtle and insidious errors. Thus, they provide no guarantee that whatever happens in the environment, your software will be impervious to attack and is not going to crash.

Some software bugs are very subtle. They may be triggered by use cases not envisioned by test planners. Such bugs often remain latent until triggered by an unexpected event... or until exploited by an industrious and ill-intentioned hacker.

Traditional software testing has failed many companies when it comes to security from cyberattacks. Here are just two recent examples:

In January 2017, the US FDA and Dept. of Homeland Security issued warnings against at least 465,000 St. Jude's Medical RF-enabled cardiac devices. Software vulnerabilities in the devices could have allowed hackers to remotely access a patient's implanted device and disable therapeutic care, drain the battery, or even administer painful electric shocks. These flaws had been revealed previously by short-selling firm Muddy Waters and the security firm MedSec, alleging negligence in St. Jude Medical's software development practices. [4]

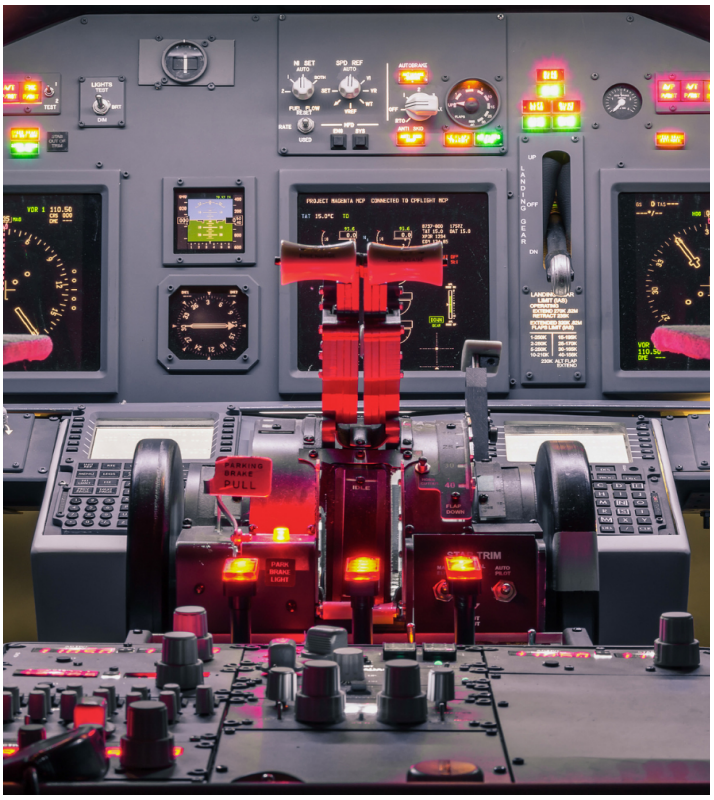
The WannaCry ransomware attack of May 2017 encrypted the data of more than 200,000 computers across 150 countries. WannaCry was based on EternalBlue, a sophisticated cyberattack exploit stolen from the U.S. National Security Agency (NSA). EternalBlue exploits a vulnerability in Microsoft's implementation of the Server Message Block (SMB) protocol in Windows and Windows Server. One month later, the NotPetya malware attack used EternalBlue to destroy data on computers across Europe, the U.S., and elsewhere. According to Kaspersky Labs, damage estimates from WannaCry range from $4 billion to $8 billion. Losses from NotPetya may have surpassed $10 billion. [5]

Why safety-critical industries are going beyond traditional software testing

Besides lacking a guarantee, traditional software has also proven to be a very expensive method for validating critical software—software that needs to be flawless, like a TEE.

In the aerospace and nuclear energy industries in the late 20th century, when traditional testing was the only available software validation option, software engineers spent much of their time reviewing their code by hand. They were keenly aware that an insidious bug might cost lives.

Since there were no tools that could guarantee their code would behave correctly, they combed through it line by line, looking for any coding irregularity that might result in an undefined behavior. It was tedious. It took lots of time and energy and was extremely costly. Software engineers were stressed and got fed up. The turnover rate among them was very high, which added to the cost.



A study by the US National Research Council in 2001 found that, "The shortage of software engineers is an acute problem in the commercial and defense aerospace industry… High-potential young software engineers often leave for jobs in non-defense industries where pay scales are higher and perceived opportunities are more exciting." [6] Burnout from code review may have been a contributing factor as well.

By the end of the 1990s, it had become clear in safety-critical industries that software that had to be perfect needed new verification tools that could guarantee determinism and safety. Now, twenty years later, with the mass proliferation of both web-connected devices and cybercrime, it's clear that similar tools are needed in the development of trusted execution environments, as well as other applications tasked with protecting connected devices from cyberattacks.

The allure of formal methods

At the beginning of this century, aerospace and nuclear engineers began looking toward formal methods. They spoke with researchers in the field. Their biggest question: "Could you help us actually guarantee the behavior of our code?"

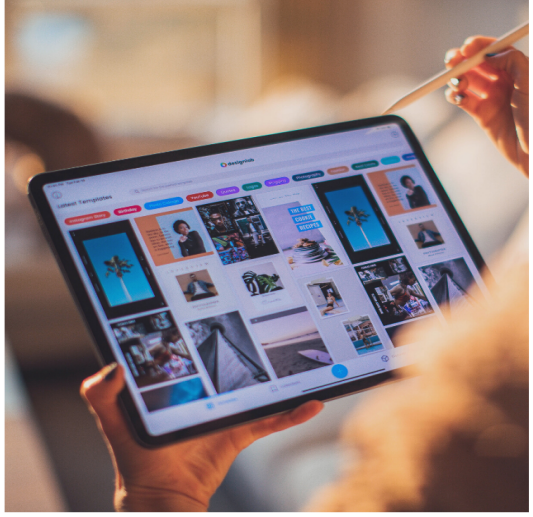
Formal methods use mathematical techniques to "solve" the logic of computer programs or other systems (integrated circuits, for example) to answer questions about their behavior. For example, if you want to know if there is any way a buffer overflow could occur in your program, formal methods can be used to determine that. What's more, a state-of-the-art formal methods tool can answer those questions for you automatically.

Formal methods are ideal for validating code that needs to be perfect. A sound formal methods tool will report every defect in your code.

That may not sound so useful if you're not worried about users finding bugs that can be fixed in a future release. But it's highly useful if you're developing a TEE that needs to be 100% reliable and impervious to cyberattacks.

Sound formal methods tools do more than find bugs. When you do traditional testing, you can never be certain you didn't forget an important test case. Formal methods, as we'll see later, can guarantee you have exhaustive test coverage of your requirements.

Formal methods can also be used to guarantee your program exactly matches its specification. That doesn't guarantee your application will work perfectly; there may be errors in the requirements. But your task then becomes reviewing the specification. That's far easier than reviewing the full source code, as those poor aerospace-and-nuclear-sector software engineers used to do.


The limitations of model checking

Widespread use of formal methods began in the semiconductor industry in the 1990s. Following the discovery of the Intel Pentium bug, Intel and the rest of the semiconductor industry began using a formal method called model checking to guarantee the functionality of their microprocessors before release. [7]

To use model checking, you first construct a model of your system. Then, you check the behavior of your model against its specification. Both the model of the system and its specification have to be formulated in some precise mathematical language. Model checking works fine when components are hardwired together, as when you have physical connections between transistors in a microprocessor. This method is still used in the semiconductor industry today.

Once you add software to your system, however, the situation becomes much more complicated. In software, rather than always having fixed connections between components, you can use pointers to address a vast number of memory addresses.

The conceptual model for a microprocessor running software is much more versatile than that of the hardware itself, because at any point in the program you can point to any memory location.

That ability for data or program flow to go anywhere makes it far more difficult to guarantee a system's behavior. That's why it's so challenging to validate a TEE: a TEE can be just as complicated as an operating system. Very often, it is indeed an operating system in itself.

What's more, programming styles can vary significantly within a TEE.

The separation kernel which forms the basis of the TEE is written in very low-level code. It typically runs on bare metal, interfacing directly with the hardware. In contrast, the trusted applications within the TEE are written in high-level style. They're similar to typical applications, but they need to be completely bug-free. In between is a communication layer, such as VPN, which is written at a level somewhere in between that of the separation kernel and the trusted apps.

This variety of programming styles, along with system complexity, makes it impossible to validate a TEE using only a single formal method like model checking.

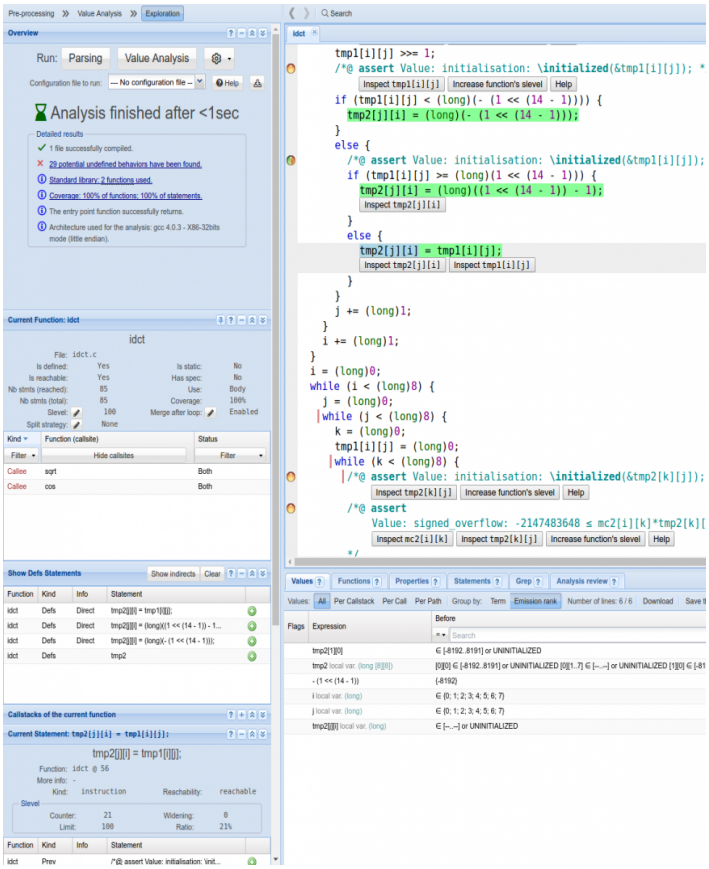
Fortunately, research in formal methods and development of formal methods tools has evolved significantly since the mid-1990s. There now exist solutions that automatically choose and combine a full range of formal methods to find and eliminate software defects and formally validate software-based systems like TEEs.

Plus, in most cases, the selection and application of these formal methods is completely transparent to the user. You don't need to be a formal methods expert. Any software developer can learn how to use these tools quickly and easily.

Exhaustive static analysis – guaranteeing trust in your trusted execution environment

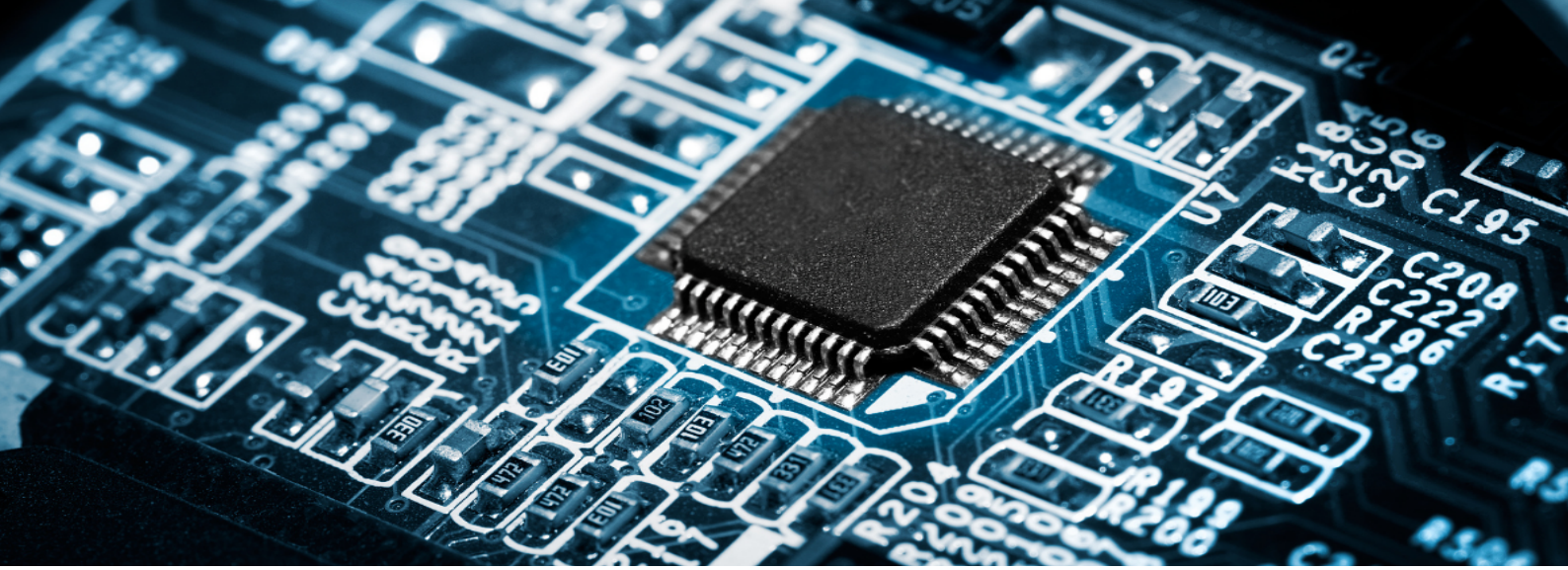
The key to guaranteeing the trustworthiness of your TEE is exhaustive static analysis.

Exhaustive static analysis is a methodology that makes use of a variety of formal methods techniques to answer the questions users ask about their code. It takes know-how developed to guarantee the behavior of safety-critical systems and expands its scope to guarantee data security as well as safety and functionality. It makes that know-how available to all developers for use in the development of any device—from smartphones to game consoles, medical technology to remote monitoring devices.



Exhaustive static analysis is also a framework where a broad range of formal methods collaborate together. It contains algorithms which choose the right formal method according to the question being asked. And it can switch seamlessly from one formal method to another, depending on the problem it has been asked to solve.

In other words, exhaustive static analysis is a holistic approach, not one of brute force. Rather than trying to solve every problem using a single formal method, the framework has been designed to determine which formal method or combination of formal methods is best suited to solving the problem at hand and to apply those methods in the best way possible.



Plugs right into your existing development process

What's more is that exhaustive static analysis does all this in a way that does not disrupt the normal software development process. It brings the value of mathematical formal methods to software development in a way that's practically invisible to the developer.

With an exhaustive static analysis framework, all the aforementioned selection and application of formal methods are totally transparent to the developer. From a user's perspective, they are simply testing their code in a manner very similar to what they're already accustomed.

The framework expands and automates the testing process. For example, by adding just a couple of lines of code to your test script, you can expand a limited set of test cases to a complete set of test cases covering the complete range of possible values for all input variables.

Again, you don't need a PhD in formal methods to use these tools. Any developer can handle them. Users don't even need to be aware of the method the tool is using.

They just express the problem and the tool does the analysis automatically. It's extremely easy for users to find answers. The framework chooses the right tools for the problem and switches tools on the fly.

For example, our own platform, TrustInSoft Analyzer for C and C++, employs several abstract memory models to analyze the software being validated. This process is completely hidden from users. They needn't know anything about them. TrustInSoft Analyzer has been designed and developed so that no matter how a C or C++ developer programs, it will extract the meaning of the code so that the right formal methods are applied. It was cited in a National Institute of Standards and Technology report to the White House[8] for having demonstrated that it can be used to formally guarantee that (1) no known undesired behaviors (bugs) are present in a system and that (2) the system behaves exactly according to its specification.[9,10]

The major benefits of exhaustive static analysis

Using exhaustive static analysis, you can guarantee the absence of coding flaws—like buffer overflows—that hackers exploit. That means it's now possible to end the cat-and-mouse game of hackers finding vulnerabilities and developers scrambling to remove them. With exhaustive static analysis, you can eliminate all such flaws before hackers even have the chance to look for them.

In fact, you'll eliminate many flaws conventional tools fail to catch. For example, using TrustInSoft Analyzer, existing test scripts downloaded from Github and a fuzzer, we recently uncovered thirteen undefined behaviors (UB) in the popular, open-source network protocol analyzer and IP packet sniffer Wireshark (Figure 2).[11] These flaws had been in the Wireshark code for years, left undiscovered by other tools.

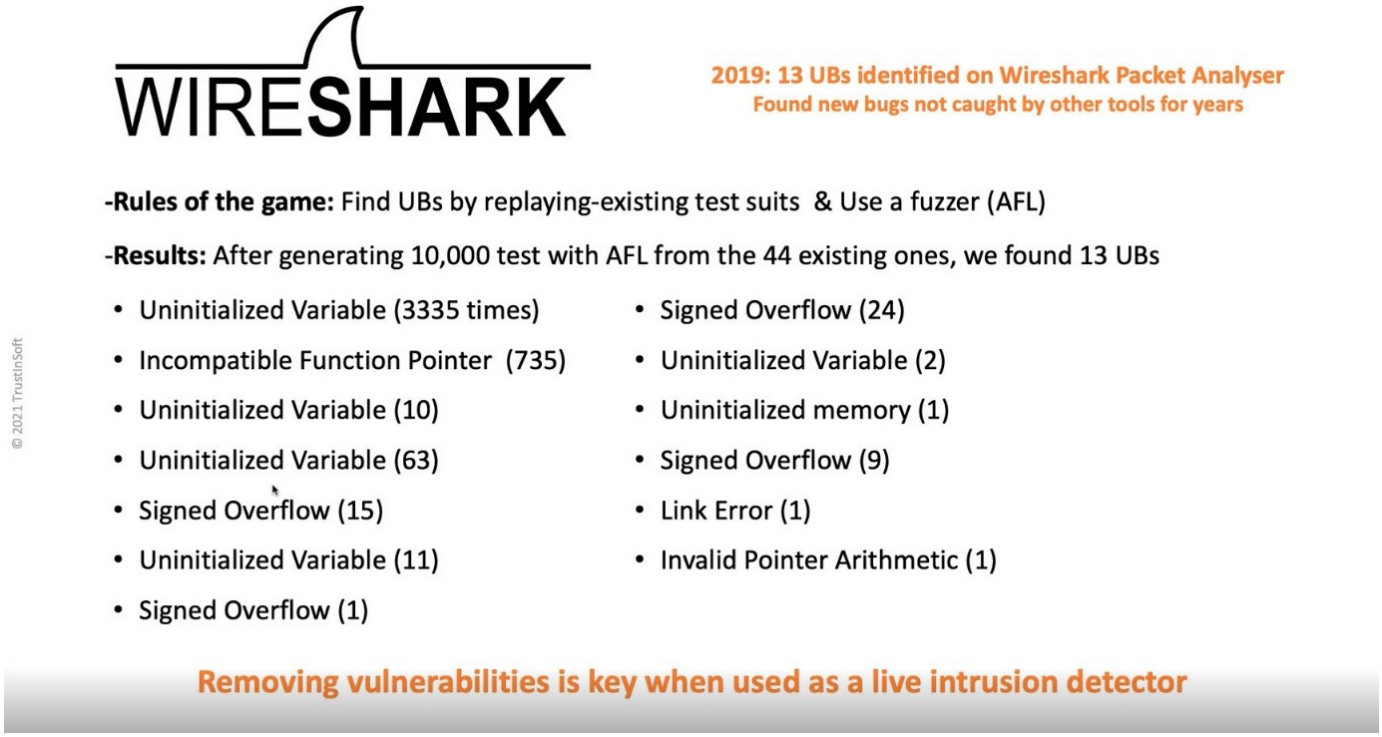


Figure 2: Previously undetected bugs uncovered by exhaustive static analysis of Wireshark

Then, if you want to go a step farther, you can guarantee your code behaves exactly in accordance with its specification—exactly what the NIST told the White House.

Exhaustive static analysis enables companies to guarantee to customers and regulatory bodies that their applications are completely free of coding flaws and conform exactly to their functional requirements. These guarantees are critical in a TEE. One other benefit of exhaustive static analysis: developer satisfaction.

Exhaustive static analysis helps developers perfect the applications they've worked so hard to create. It helps them quickly find and eliminate those subtle coding errors that are so easy to make without even being aware of them… and that are so hard to track down after they've been made. It saves developers time. It makes them feel great about what they've built. Plus, it does all this in an automated, fully comprehensive manner. Exhaustive static analysis finds all the needles in your haystack.

Exhaustive static analysis at work on TEEs – two case studies

Case study #1: Cybersecurity application developer

A cybersecurity company wanted to dynamically analyze the presence and behavior of malware in a system. To do that, they needed to run the malware in a controlled environment—a dangerous undertaking, as that functioning malware could have invaded and harmed their own systems if not perfectly contained.

The company developed a TEE to control how they were running those viruses to make sure their systems would not be contaminated. They then formally verified their TEE using TrustInSoft Analyzer to guarantee it is free of all vulnerabilities and behaves exactly as specified. They can now do their analysis in confidence, knowing any malware they're analyzing is reliably contained.

Case study #2: Communication device manufacturer

This large company makes a wide range of devices and manufactures thousands of each model. Like most of their competitors, they rely on a TEE to protect their devices and their customers' sensitive information for applications like fingerprint recognition, facial recognition, mobile payments, and so forth.

Needless to say, a hack of their TEE could affect millions of devices and users. That manufacturer, too, has chosen TrustInSoft Analyzer to formally verify its TEE.

What to look for when choosing an exhaustive static analysis solution

Choosing the exhaustive static analysis solution that's right for your organization can be challenging. Here are three important features to look for:

1. Applies a wide range of formal methods in a manner transparent to the user

- The field of formal methods covers a variety of methods that are used to solve problems of logic and mathematics. Each method was created and is best used for a specific type of problem. A good exhaustive static analysis tool will be able to apply a wide range of these methods to solve a wide variety of problems. Only then can it hunt down the full range of possible undefined behaviors, and guarantee the security and functionality of your TEE. Furthermore, selection and application of specific formal methods should be done automatically by the tool without the need for user intervention. Scientists in the field of formal methods spend years learning how to apply these complex techniques. Your developers shouldn't have to.

2. Fits seamlessly into your current development process

- Adopting an exhaustive static analysis tool shouldn't disrupt your current development process or even cause you to alter it significantly. Using the tool should be similar to the experience you now have using conventional software testing tools.

3. Offers several hierarchical levels of analysis

- Tracking down and eliminating undefined behaviors and guaranteeing the security and reliability of a TEE is an iterative, cumulative process. The process benefits from taking a step-by-step hierarchical approach, stepping from a basic level of proof to more advanced levels, depending on what a given application requires.
  For example, TrustInSoft Analyzer offers three levels of analysis.

LEVEL 1

- Level 1 (testing) is the simplest to use. It is completely automated—as easy to use as ordering a compile of your code. The user doesn't even have to look at the code to use it. Yet, Level 1 will uncover a large portion of the bugs present in the code without yielding any false positives. The Wireshark analysis mentioned earlier was done exclusively with Level 1. Level 1 is ideal for use in a continuous integration process. New bugs can be stripped out on a daily basis before they accumulate, without developers having to look for them.

LEVEL 2

- (exhaustive testing) will guarantee that all coding defects have been eliminated from your code—that you have no undefined behaviors present that can be exploited by hackers. Use of Level 2 requires some operator intervention, but it has a very low false positive rate (<10/10,000LOC on average). Level 2 provides you with a guarantee of the security of your system if all the confirmed defects are corrected.

LEVEL 3

- Finally, for those applications that need it, there is Level 3 (functional proof). Level 3 guarantees your code fulfills its specification exactly. Functional proof takes longer and is more costly than defect testing, but for applications that need to be perfect—like a TEE, for example—it is well worth it. The return is huge. You have a guarantee that your critical application works exactly as specified.

Conclusions

A trusted execution environment needs to work securely and flawlessly. Traditional software testing methods are inadequate for the task of TEE development and offer no guarantee of security or reliability.
Exhaustive static testing, on the other hand, can provide guarantees that your TEE is secure, 100% free of defects that hackers can exploit, and behaves exactly as specified.

About TrustInSoft

TrustInSoft participates in the Application Security Testing market alongside vendors such as Mathworks, Parasoft, Synopsys and Veracode. TrustInSoft Analyzer is a hybrid static and dynamic code analyzer that automates Formal Methods to mathematically guarantee C/C++ code quality, security and safety. TrustInSoft has customers worldwide in the automotive, IoT, telecom, semiconductor, aeronautics and defense industries. The company received awards and recognition from NIST, RSA and Linux Foundation.

To learn more about TrustInSoft Analyzer, visit trust-in-soft.com/product-c-and-c-source-code-analyzer/.

If you'd like to speak with a TrustInSoft technical representative about how TrustInSoft Analyzer can meet your organization's specific needs, contact us by email at contact@trust-in-soft.com.

References

1 Prado, S.; Introduction to Trusted Execution Environment and ARM's TrustZone; #embeddedbits, March 2020.

2 Ibid.

3 Assaraf, A.; This is what your developers are doing 75% of the time, and this is the cost you pay; Coralogix, February 2015.

4 465,000 Abbott pacemakers vulnerable to hacking, need a firmware fix; CSO, September 2017.

5 Snow, J.; Top 5 most notorious cyberattacks; Kaspersky, December 2018.

6 National Research Council; Review of the Future of the U.S. Aerospace Infrastructure and Aerospace Engineering Disciplines to Meet the Needs of the Air Force and the Department of Defense; The National Academies Press, 2001.

7 Clarke, E. M., Khaira M., Zhao, X.; Word Level Model Checking – Avoiding the Pentium FDIV Error; Proceedings of the 33rd Annual Conference on Design Automation Conference – DAC '96. DAC '96: 645–648.

8 Black, P.; Badger, L.; Guttman, B.; Fong, E.; Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy; National Institute of Science and Technology (NIST), November 2016.

9 Bakker, P.; Providing assurance and trust in PolarSSL; May 2014

10 Regehr, J.; Comments on a Formal Verification of PolarSSL; https://blog.regehr.org, September 2015.

11 Zupkus, A; The Wireshark Challenge: how to ensure the security of open-source projects with formal methods; TrustInSoft, May 2020.