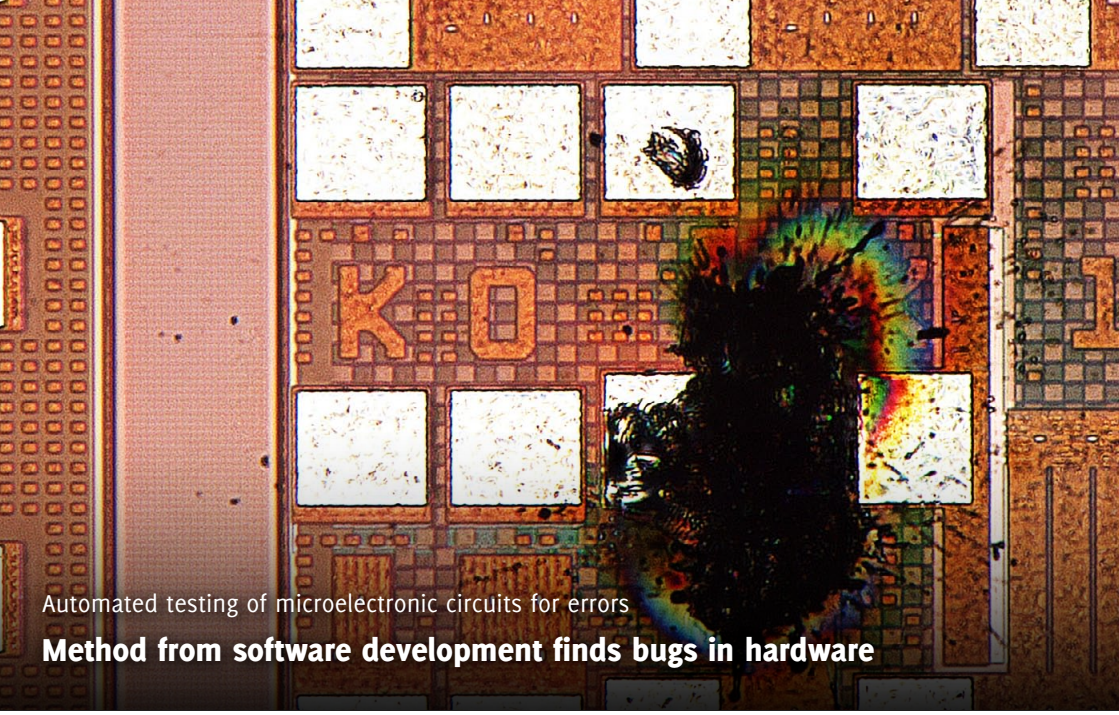

Automated testing of microelectronic circuits for errors

Method from software development finds bugs in hardware

If errors are made in the design of microelectronic chips and not detected in time, the consequences can be catastrophic, for example in network technology, automotive electronics, avionics or pacemakers. The image shows an example of destroyed wafer-level components. Source: IMMS.

Motivation and overview

Verification of integrated circuits is an essential part of the design process and presents unique challenges. The ever-increasing complexity of integrated circuits is being achieved by increasing automation on the design side, and verification must keep pace. Despite partially automated methods such as constrained random testing, an experienced verification engineer is needed to develop targeted test scenarios and verify system states.

In the area of software, so-called “fuzz testing” has become established, which floods the software to be tested with randomly generated inputs and can thus uncover errors in the functioning of the programmes. Inspired by this verification method, IMMS has developed a method in the KI-EDA project that can automatically test microelectronic circuits for bugs and vulnerabilities. For this purpose, the circuit is confronted with randomly generated inputs that are varied step by step by genetic algorithms in order to provoke faulty and unsafe system behaviour. The random nature of the generated inputs makes this type of search particularly suitable for finding faults that

www.imms.de/ki-eda

are not found by typical human-generated test scenarios. The method can be used to augment established verification techniques and assist verification engineers with autonomous fault finding.

Background and challenges

Increasing complexity of integrated circuits

Since the first integrated circuit about 65 years ago, microchips have been a driving force behind technological development. While Jack Kilby's first chip in 1958 consisted of a single flip-flop, modern processors contain billions of components. The complexity of these digital circuits has replaced "rocket science" as the epitome of sophisticated engineering that is hard to keep track of.

In the beginning, circuits were designed by hand, but today engineers rely on specialised electronic design automation (EDA) software. Even the "little brother" of the complex high-performance processor, the application-specific integrated circuit (ASIC), contains logic blocks that are getting bigger and bigger, giving us the functions we are familiar with today: WiFi, Bluetooth, high data throughput. This "higher, faster, further" of chip development is made possible on the design side by the high degree of automation without which successful and, above all, safe chip development is no longer conceivable.

www.imms.de/eda

Verification of complex integrated circuits has so far been largely experience-based and manual

Again, safety is the key word. While EDA is advancing design automation with amazing methods, complex and difficult-to-verify circuits remain a major challenge for quality control and management. Verification engineers must not only ensure that the circuit can perform its intended function. They must also ensure that unforeseen conditions do not occur during operation and that design errors are avoided. If the camera in the smart refrigerator fails because of a system failure, it can be annoying. But with network technology, digital steering controls in cars, or pacemakers, system failures can be catastrophic.

However, verification has become a bottleneck in the ASIC design process because it has been difficult to automate the human experience and intuition required for efficient debugging. To solve this problem, and to support verification by at least partially automating debugging, it is worth looking beyond one's own nose - to so-called fuzzing, which has been successfully used in the field of software verification for years.

Drawing inspiration from software development

Basic principle of software-fuzzers – random inputs find random bugs

For software, the basic principle of fuzzing is quite simple and sounds banal at first: the application under test is bombarded with random inputs, without any detailed knowledge of the correct syntax or structure, in order to crash it. It is then possible to analyse how the inputs caused the error and how this can be avoided in the future. This approach is already used effectively to find security vulnerabilities in software. One of the most widely used software fuzzers is American Fuzzy Lop (AFL), which already has an impressive trophy collection: In addition to finding bugs in web browsers like Mozilla Firefox or Internet Explorer, it also finds hits in programmes like Adobe Reader and MySQL, as well as operating systems like Linux or iOS.

The great advantage of fuzzers is their high degree of automation. Once started, fuzzers run without further intervention until they find something or the search is aborted. For larger projects, such as the Linux kernel, multiple fuzzers are sent out on a parallel search over several days or weeks.

The idea of using this approach to verify electronic circuits is obvious. However, several approaches to integrate fuzzers into hardware verification have emerged only recently, e.g., [1].

Pure randomness does not work – the Infinite Monkey Theorem

In fact, fuzzing is not that simple. Rather, the idea that if you randomise the inputs to a system long enough, something meaningful will come out is reminiscent of the famous Infinite Monkey Theorem. It was originally stated by the French math-

ematician Émile Borel in 1913. He claimed that if a million trained monkeys each sat at a typewriter for 10 hours a day and typed randomly, they would produce the equivalent of the entire world's literature within a year. While the probability of this event occurring is not zero, it can be shown to be so small as to have no practical relevance. For fuzzing, this means that it is too unlikely that a random sequence of bits corresponds to, say, a corrupt JPEG file that causes an error in the software under test.

Building upon close misses – game principle of "Battleship" helps to find the bullseye

To solve this problem, most fuzzers like AFL use an approach called coverage-guided fuzzing. The underlying assumption is that a random input must also cause a response in the programme under test in order to provoke an error. Typically, software applications do not respond to input noise, but require certain file formats or communication protocols to start execution. If the programmes are not executed, they will not be able to exhibit the erroneous behaviour that is being sought. Programme coverage provides information about which parts of a programme were actually executed. Noise on the inputs of a software application is very likely to result in low coverage, while the input of, for example, a file with the correct file format will cover the execution of the corresponding function in the programme.

The information about the achieved programme coverage triggered by an input is used by fuzzers to avoid inputs that cannot cause an error. The problem can be illustrated by the game "Battleship". On an unknown and invisible playing field, a ship is hidden over several squares, and the player's task is to find and sink it. When the player announces the coordinates of a square, the opponent tells him whether he has hit it or not. The player now tries to hit the ship once by randomly calling out the coordinates, and upon a hit he tries to search the neighbouring squares and sink the ship by hitting the rest of the corresponding squares. If coverage-guided fuzzers happen to find an input that leads to a slightly larger programme coverage than simple noise, they try to search the "neighbouring" space by slightly modifying or mutating the original input.

Since each programme input is given a value for programme coverage, these values can be viewed as a landscape on a hyperplane in the space spanned by the inputs.

However, unlike common optimisation algorithms, the fuzzer here does not try to find local maxima, but rather searches for the peaks while avoiding the flat planes. Due to the similarity of the problem, "coverage-guided mutation-based" fuzzers, such as the AFL mentioned above, use algorithms from the fields of optimisation and genetics to search the input space and generate new inputs through mutations.

Fuzz-testing and chip design

Challenge: known approaches to transfer software fuzzers unsuitable for mixed-signal circuits

If fuzzing can be used effectively to debug complex software, why wouldn't it work for hardware development? Unlike software development, chips cannot be tested directly during design because they often do not yet exist as physical circuits. Therefore, they have to be emulated with programmable logic devices (FPGA) or simulated on the computer with appropriate software. This not only requires a lot of computing power. A hardware fuzzer also has to combine a variety of simulation software, data formats, and mathematical algorithms in a single workflow.

To address these challenges, several proposals have recently been published:

- Use software fuzzers and adapt them to hardware simulation software. This has the advantage of using already established and efficient fuzzers [1].
- Use of optimised simulation software for hardware or FPGA prototypes. This can speed up or even bypass the simulations, significantly speeding up fuzzing [2].

Unfortunately, both of these approaches have drawbacks that particularly affect the design of ASICs. Their function is often based on the interaction of analogue and digital circuit elements, which must be considered together. While optimised hardware simulators and FPGA prototypes can only simulate or emulate purely digital circuits, mixed-signal simulations are much more complex and cannot be directly replaced by FPGA prototypes.

In addition, adapting software fuzzers for hardware verification is very costly and inflexible. Furthermore, the approach published so far is limited to pure digital circuits.

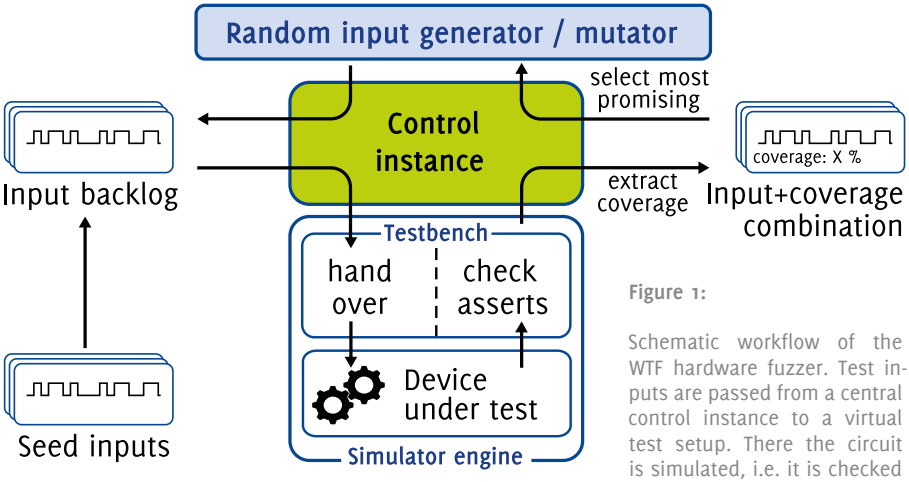

Figure 1:

Schematic workflow of the WTF hardware fuzzer. Test inputs are passed from a central control instance to a virtual test setup. There the circuit is simulated, i.e. it is checked whether the inputs can put the circuit into an undesired state. For each input, it is determined which parts of the circuit were active. With this information, new test cases are generated by mutating and selecting previous inputs. As a result, the fuzzer only generates test cases that actually trigger activity in the circuit. Source: IMMS.

„What the Fuzz“ (WTF) – the hardware fuzzer of IMMS for the development of mixed-signal chips

In the KI-EDA project, IMMS has developed a hardware fuzzer which, although it does not use proven software fuzzers, has a very modular design and is also suitable for the mixed-signal properties of ASICs (Fig. 1).

www.imms.de/ki-eda

The fuzzer consists of several interchangeable modules, which are coordinated by a central instance and cover all steps of the fuzzer:

- Input management
- Simulator
- Coverage readout
- Mutator and optimisation strategy

Like all fuzzers, WTF works iteratively. At the beginning, you have to define the unwanted error states of the circuit you want the fuzzer to search for. This can be, for example, the requirement that the circuit must not be in read and write mode at the same time.

A hardware simulation analyses the behaviour of the circuit for a particular input, checks for fault conditions, and then determines which parts of the circuit were ac-

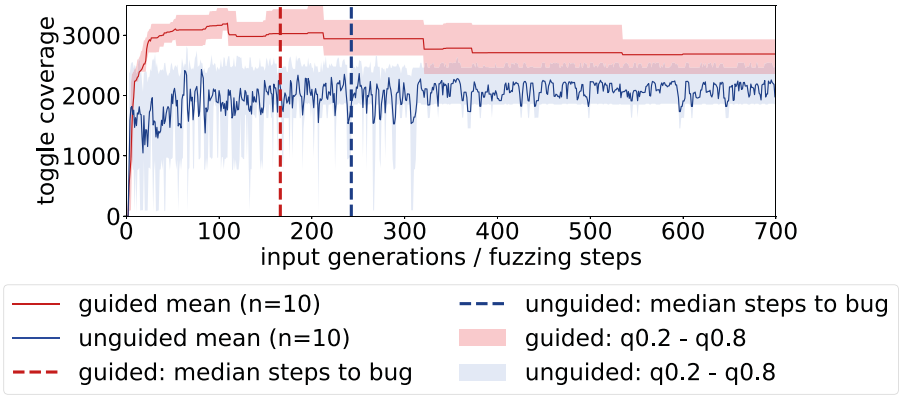

Figure 2: In this example, the fuzzer is presented with a sample circuit with an intentional fault built in to measure how quickly the fuzzer can find the fault. The data contains 10 runs of the fuzzer, each run automatically generating hundreds of test cases. Compared to unguided fuzzing (blue) with purely randomly generated test inputs, the guided fuzzer (red) finds the hidden defect much faster on average (dashed lines). It does not generate completely new inputs but mutates and selects previous ones. This allows it to generate evolutionary test cases that activate larger parts of the circuit (higher coverage), increasing the chance of finding the bug in each case. Source: IMMS.

tive during that input. This is the coverage achieved by that input. According to an optimisation strategy, a mutation is selected and a new input is generated. This new input is passed to a hardware simulation of the circuit, which checks for errors and determines the coverage achieved. To generate the next input, the two coverages achieved so far are compared. If the coverage was reduced by the last mutation, it is rejected and a new input is generated based on the original input. If the coverage was increased by the last mutation, the new input is mutated to generate the next generation of the input.

Whether mutations are accepted or not is decided by the optimisation strategy and can be stochastic, e.g. in a heuristic approximation method like simulated annealing. Typical genetic methods are used as mutations, which are easy to illustrate, especially for digital inputs. For example, input 1011 can be modified by replacing a single bit (e.g., 0011), or a position can be truncated (101), appended (10110), or inserted (10101).

Currently, WTF supports Cadence Xcelium and Icarus Verilog simulation software. However, due to the modular design of the fuzzer, the feature set can easily be extended. In particular, new optimisation strategies can be integrated.

The WTF hardware fuzzer is still in the development and testing phase, but has already proven its functionality on sample circuits [3]. The advantage of "coverage-guided fuzzing" over "unguided fuzzing", where the test inputs for the circuit under test are generated completely randomly, is well demonstrated (Fig. 2). In an experiment, WTF was tested on a sample circuit in which a defect was intentionally introduced at IMMS. Thanks to the feedback of the coverage achieved by the test inputs, WTF can efficiently generate test cases with higher coverage and thus find the fault hidden in the circuit faster.

Now that the fuzzer has demonstrated the general applicability of the fuzzing concept in the field of hardware verification, it is tested with industrial chips developed at IMMS. Approaches published by other research groups will now also be tested in practice. Since "coverage-guided fuzzing" has successfully established itself as a verification method in the software domain, it will be exciting to see whether it will also succeed in the hardware domain.

Contact person: Henning Siemen, M.Sc., henning.siemen@imms.de

Literature:

[1] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," 2021. [Online]. Available: https://arxiv.org/abs/2102.02308

[2] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in International Conference on Computer-Aided Design (ICCAD), 2018, pp. 1–8.

[3] H. Siemen, J. Lienke, and G. Gläser, "Hot Fuzz: Assisting verification by fuzz testing microelectronic hardware", in 20th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), 2023

SPONSORED BY THE

Federal Ministry of Education and Research

The KI-EDA project is funded by the Federal Ministry of Education and Research within the framework of the programme "Microelectronics for Industry 4.0 (ElektroniK I4.0)" under the consortium number es2eli4001, IMMS under the reference 16ME0010. Partners are Centitech GmbH (FRABA Group) and iC-Haus GmbH.

Automatisiertes Testen mikroelektronischer Schaltungen auf Fehler

Methode aus der Softwareentwicklung findet Bugs in Hardware

Werden beim Entwurf von Mikroelektronik-Chips Fehler gemacht und diese nicht rechtzeitig entdeckt, kann das katastrophale Auswirkungen haben, etwa bei Netzwerktechnik, digitaler Lenksteuerung im Auto oder Herzschrittmachern. Das Bild zeigt ein Beispiel für zerstörte Bauelemente auf Wafer-Ebene. Quelle: IMMS.

Motivation und Überblick

Die Verifikation von Mikroelektronik-Chips ist ein wesentlicher Teil des Entwurfsprozesses und bringt besondere Herausforderungen mit sich. Die stetig steigende Komplexität der integrierten Schaltungen wird durch eine wachsende Automatisierung auf Entwurfsseite erreicht, mit der die Verifikation Schritt halten muss. Trotz teilautomatisierter Verfahren wie „constrained random testing“ bedarf es eines erfahrenen Verifikationsingenieurs, der gezielt Testszenarien entwickelt und Systemzustände prüft. Im Softwarebereich hat sich sogenanntes „fuzz testing“ etabliert, das zu testende Software mit zufällig generierten Eingaben überschüttet und dabei Fehler in der Funktion der Programme aufdecken kann. Inspiriert von diesem Verifikationsverfahren hat das IMMS im Projekt KI-EDA eine Methode entwickelt, die automatisiert mikroelektronische Schaltungen auf Bugs und Schwachstellen prüfen kann. Hierzu wird die Schaltung mit zufällig generierten Eingaben konfrontiert, die Schritt für Schritt durch genetische Algorithmen variiert werden, um fehlerhaftes und unsicheres Systemverhalten zu provozieren. Durch die zufällige Natur der generierten Eingaben ist diese Art der Suche besonders geeignet, Fehler zu finden, die von typischen, von Menschen

www.imms.de/ki-eda

erzeugten Testszenarien nicht gefunden werden. Mit der Methode lassen sich etablierte Verifikationsverfahren ergänzen und Verifikationsingenieure durch autonome Fehlersuchen unterstützen.

Hintergrund und Herausforderungen

Steigende Komplexität integrierter Schaltungen

Seit der ersten integrierten Schaltung vor rund 65 Jahren sind Mikroelektronik-Chips ein treibender Motor der technischen Entwicklung. Während der erste Chip von Jack Kilby 1958 noch aus einem einzelnen Flipflop bestand, enthalten moderne Prozessoren mehrere Milliarden Bauelemente. Die Komplexität dieser digitalen Schaltungen hat mittlerweile die „Raketenwissenschaft" als Inbegriff anspruchsvoller und schwer zu überblickender Ingenieurswissenschaft abgelöst.

Wurden anfangs die Schaltungen noch per Hand entworfen, so müssen sich die Ingenieure heutzutage auf spezialisierte Software für den Entwurf von Elektronik per Electronic Design Automation, EDA, verlassen. Auch beim „kleinen Bruder" des komplexen Hochleistungsprozessors, der anwendungsspezifischen integrierten Schaltung (application-specific integrated circuit, ASIC), werden an Größe zunehmende Logikblöcke verbaut, die uns die mittlerweile vertrauten Funktionen bringen: WiFi, Bluetooth, hoher Datendurchsatz. Dieses „Höher, Schneller, Weiter" der Chipentwicklung wird auf Entwurfsseite durch den hohen Grad an Automatisierung ermöglicht, ohne den eine erfolgreiche und vor allem sichere Entwicklung von Chips mittlerweile undenkbar ist.

www.imms.de/eda

Verifikation komplexer integrierter Schaltungen bislang größtenteils erfahrungsbasiert und von Hand

Sicherheit ist hier auch das richtige Stichwort. Denn während die EDA mit erstaunlichen Methoden die Entwurfsautomatisierung vorantreibt, so bleiben die komplexen und schwer zu überschauenden Schaltungen eine große Herausforderung für Qualitätskontrolle und -management. Verifikationsingenieure müssen nicht nur sicherstellen, dass die Schaltung ihre geplante Funktion erfüllen kann. Sie müssen auch darauf achten, dass während des Betriebs keine unvorhergesehenen Zustände eintreten können und Entwurfsfehler vermieden werden. Wenn die Kamera im smarten Kühlschrank durch einen Systemfehler ausfällt, mag das ärgerlich sein. Bei Netzwerktechnik, digitaler Lenksteuerung im Auto oder Herzschrittmachern hingegen können Systemfehler katastrophale Auswirkungen haben.

Die Verifikation hat sich allerdings im Entwurfsprozess von ASICs zum Flaschenhals entwickelt, da es bisher nur schwer möglich war, die zur effizienten Fehlersuche benötigte menschliche Erfahrung und Intuition zu automatisieren. Um dieses Problem zu lösen und die Verifikation wenigstens durch eine Teilautomatisierung der Fehlersuche zu unterstützen, lohnt sich ein Blick über den Tellerrand – zum sogenannten Fuzzing, das im Bereich der Softwareverifikation schon seit Jahren erfolgreich eingesetzt wird.

Lösungsansätze aus der Software-Entwicklung

Grundprinzip von Software-Fuzzern – zufällige Eingaben decken Fehler auf

Für Software ist das Grundprinzip des Fuzzing denkbar einfach und klingt zunächst banal: die zu testende Anwendung wird ohne genauere Kenntnis von korrekter Syntax oder Struktur mit zufälligen Eingaben bombardiert, um sie zum Absturz zu bringen. Anschließend kann man analysieren, wie die Eingaben den Fehler erzeugt haben und wie dies zukünftig vermieden werden kann. Dieser Ansatz wird bereits effektiv genutzt, vor allem, um sicherheitsrelevante Schwachstellen in Software zu finden. Einer der meistgenutzten Software-Fuzzer ist American Fuzzy Lop (AFL), der bereits auf eine beeindruckende Trophäensammlung blicken kann: Neben gefundenen Fehlern in Webbrowsern wie Mozilla Firefox oder dem Internet Explorer reihen sich dort auch Treffer in Programmen wie dem Adobe Reader und MySQL sowie Betriebssystemen wie Linux oder iOS ein.

Der große Vorteil von Fuzzern ist der hohe Automatisierungsgrad. Einmal gestartet, laufen Fuzzer ohne weiteres Zutun, bis sie etwas gefunden haben oder die Suche abgebrochen wird. Für größere Projekte wie dem Linux Kernel werden mehrere Fuzzer parallel für mehrere Tage oder Wochen auf die Suche geschickt.
Der Gedanke, diesen Ansatz auch für die Verifikation von elektronischen Schaltungen zu verwenden, liegt nahe. Es gibt jedoch erst seit kurzem verschiedene Ansätze, Fuzzer in die Hardwareverifikation zu integrieren, z.B. [1].

Der Zufall hilft aber oft nicht weiter – das Infinite-Monkey-Theorem

Tatsächlich funktioniert Fuzzing dann doch nicht so einfach. Vielmehr erinnert die Idee, dass wenn man nur lange genug zufällig an den Eingängen eines Systems herumprobiert, auch etwas Vernünftiges dabei herauskommt, an das berühmte Infinite-Monkey-Theorem. Es wurde ursprünglich vom französischen Mathematiker Émile Borel 1913 aufgestellt. Dieser behauptete, dass eine Million dressierter Affen, die

10 Stunden täglich an Schreibmaschinen sitzen und auf ihnen zufällig herumtippen, innerhalb eines Jahres eine Textpassage erzeugen, die der gesamten Weltliteratur entspricht. Die Wahrscheinlichkeit, dass dieses Ereignis eintritt, ist zwar nicht Null, es lässt sich jedoch zeigen, dass sie so gering ist, dass sie keinerlei praktische Relevanz hat. Für das Fuzzing bedeutet dies, dass es zu unwahrscheinlich ist, dass eine zufällige Abfolge von Bits beispielsweise einer defekten JPEG-Datei entspricht, die einen Fehler in der zu testenden Software auslöst.

Aus Fehlern lernen – Spielprinzip „Schiffe versenken“ provoziert die „richtigen“ Fehler

Um diesem Problem zu begegnen, verwenden die meisten Fuzzer wie AFL einen Ansatz namens „coverage-guided fuzzing“. Dahinter steckt die Annahme, dass eine zufällige Eingabe auch eine Reaktion im zu testenden Programm hervorrufen muss, um einen Fehler zu provozieren. Normalerweise reagieren Softwareanwendungen nicht auf Eingangsrauschen, sondern benötigen spezifische Dateiformate oder Kommunikationsprotokolle, um eine Ausführung zu starten. Werden die Programme nicht ausgeführt, können sie auch nicht das gesuchte fehlerhafte Verhalten zeigen. Die „Coverage“ oder Abdeckung gibt Auskunft darüber, welche Teile eines Programms auch tatsächlich ausgeführt wurden. Ein Rauschen auf den Eingängen einer Softwareanwendung führt sehr wahrscheinlich zu einer geringen Abdeckung, während die Eingabe beispielsweise einer Datei mit korrektem Dateiformat die Ausführung der entsprechenden Funktion im Programm abdeckt.

Die Information über die erreichte Programmabdeckung, die durch eine Eingabe ausgelöst wird, nutzen Fuzzer, um Eingaben zu vermeiden, die keinen Fehler hervorrufen können. Veranschaulichen lässt sich das Problem mit dem Spiel „Schiffe versenken“. Auf einem unbekannten und nicht einsehbaren Spielfeld ist ein mehrere Kästchen großes Schiff versteckt und der Spieler hat die Aufgabe, es zu finden und zu versenken. Nennt der Spieler die Koordinaten eines Kästchens, erfährt er vom Gegenspieler, ob er getroffen hat oder nicht. Der Spieler versucht nun mit zufälligen Ansagen, das Schiff einmal zu treffen um danach gezielt die benachbarten Kästchen abzusuchen und das Schiff mit Treffern auf allen zugehörigen Kästchen zu versenken. Wenn Coverage-guided-Fuzzer zufällig eine Eingabe gefunden haben, die zu einer etwas größeren Programmabdeckung als einfaches Rauschen führt, dann versuchen sie den „benachbarten“ Raum abzusuchen, indem sie die ursprüngliche Eingabe nur leicht modifizieren oder mutieren.

Da jede Programmeingabe einen Wert für eine Programmabdeckung erhält, kann man diese Werte als Landschaft auf einer Hyperebene im durch die Eingaben aufgespannten Raum betrachten. Im Gegensatz zu gängigen Optimierungsalgorithmen versucht der Fuzzer hier jedoch nicht, lokale Maxima zu finden, sondern die Berg- und Hügelkuppen abzusuchen und dabei die flachen Ebenen zu vermeiden. Aufgrund der Ähnlichkeit der Problemstellung verwenden „Coverage-guided mutation-based"-Fuzzer, wie der genannte AFL, Algorithmen aus dem Bereich der Optimierung und der Genetik, um den Eingaberaum abzusuchen und durch Mutationen neue Eingaben zu erzeugen.

Fuzzing Hardware im Chip-Entwurf

Herausforderung: bekannte Ansätze zur Übertragung von Software-Fuzzern für Mixed-Signal-Schaltungen ungeeignet

Wenn Fuzzing effektiv zur Fehlersuche in komplexer Software verwendet werden kann, warum sollte das dann nicht auch im Bereich der Hardwareentwicklung funktionieren? Im Unterschied zur Software-Entwicklung können die Chips während des Entwurfs nicht direkt getestet werden, da sie oftmals noch nicht als physische Schaltung vorliegen. Sie müssen daher aufwendig mit programmierbaren Logikbausteinen (FPGA) emuliert oder mit entsprechender Software am Computer simuliert werden. Das erfordert nicht nur umfangreiche Rechenleistung. Ein Hardware-Fuzzer muss auch eine Vielzahl von Simulationssoftware, Datenformaten und mathematischen Algorithmen in einem Arbeitsablauf miteinander verbinden.

Um diesen Herausforderungen zu begegnen, wurden zuletzt mehrere Vorschläge publiziert:

- Verwendung von Software-Fuzzern und deren Anpassung an Simulationssoftware für Hardware. Das hat den Vorteil, auf bereits etablierte und effiziente Fuzzer zurückgreifen zu können [1].
- Verwendung von optimierter Simulationssoftware für Hardware oder FPGA-Prototypen. Dadurch können die Simulationen beschleunigt oder sogar umgangen werden, was das Fuzzing deutlich beschleunigt [2].

Leider haben diese beiden Vorschläge Nachteile, die insbesondere den Entwurf von ASICs betreffen. Deren Funktion beruht oft auf dem Zusammenspiel analoger und digitaler Schaltungselemente, die zusammen betrachtet werden müssen. Während optimierte Hardwaresimulatoren und FPGA-Prototypen nur reine digitale Schaltungen simulieren bzw. emulieren können, sind sogenannte „Mixed-Signal"-Simulationen

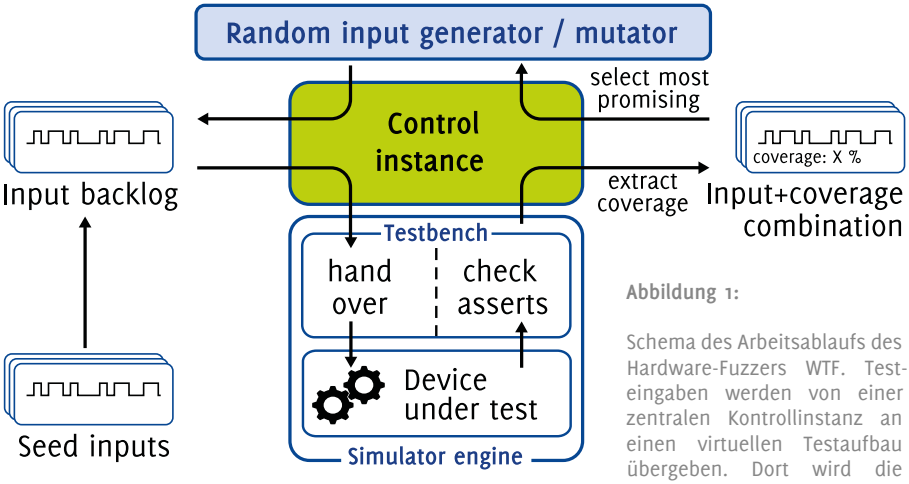

Abbildung 1:

Schema des Arbeitsablaufs des Hardware-Fuzzers WTF. Testeingaben werden von einer zentralen Kontrollinstanz an einen virtuellen Testaufbau übergeben. Dort wird die Schaltung simuliert, d.h. überprüft, ob die Eingaben die Schaltung in einen unerwünschten Zustand versetzen können. Für jede Eingabe wird ermittelt, welche Teile der Schaltung aktiv waren. Mit dieser Information werden durch Mutation und Selektion vorhergehender Eingaben neue Testfälle generiert. Dadurch erzeugt der Fuzzer nur Testfälle, die auch eine Aktivität in der Schaltung auslösen. Grafik: IMMS.

deutlich aufwendiger und können nicht direkt von FPGA-Prototypen ersetzt werden. Zudem ist die Anpassung von Software-Fuzzern für die Hardwareverifikation sehr aufwendig und unflexibel. Darüber hinaus ist der bisher publizierte Ansatz ebenfalls auf die Verwendung für rein digitale Schaltungen beschränkt.

„What the Fuzz“ (WTF) – der Hardware-Fuzzer des IMMS zur Entwicklung von Mixed-Signal-Chips

www.imms.de/ki-eda

Im Projekt KI-EDA hat das IMMS einen Hardware-Fuzzer entwickelt, der zwar auf die erprobten Software-Fuzzer verzichtet, dafür jedoch sehr modular aufgebaut ist und sich auch für die Mixed-Signal-Eigenschaften von ASICs eignet (Abb. 1).

Der Fuzzer besitzt mehrere austauschbare Module, die von einer zentralen Instanz koordiniert werden und alle Arbeitsschritte des Fuzzers abdecken:

- Verwaltung der Eingaben
- Simulator
- Auslesen der Abdeckung (Coverage)
- Mutator und Optimierungsstrategie

Wie alle Fuzzer arbeitet WTF iterativ. Zu Beginn müssen unerwünschte Fehlerzustände der Schaltung definiert werden, nach denen der Fuzzer suchen soll. Dies kann zum Beispiel die Anforderung sein, dass sich die Schaltung nicht gleichzeitig sowohl im Lese- als auch im Schreibmodus befinden darf.

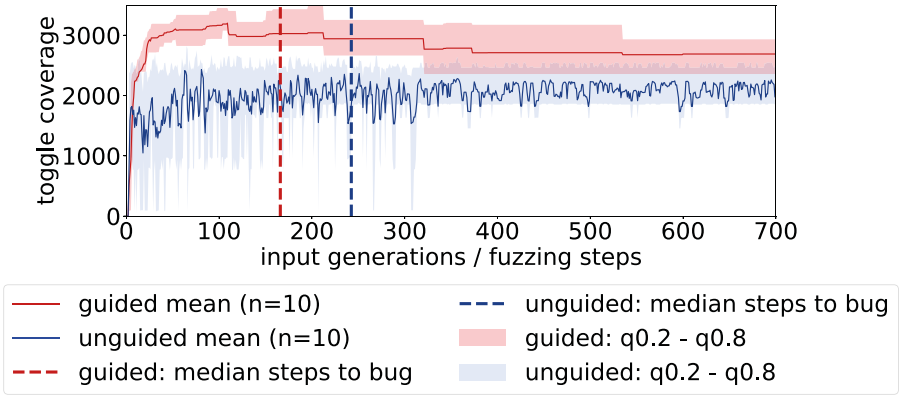

Abbildung 2: In diesem Beispiel wird dem Fuzzer eine Schaltung mit einem absichtlich eingebauten Fehler präsentiert, um zu messen, wie schnell der Fuzzer den Fehler finden kann. Die Daten beinhalten je 10 Durchläufe des Fuzzers, der jedes Mal automatisiert und nacheinander hunderte Testfälle generiert. Im Vergleich zum ungelenkten Fuzzing (Blau) mit rein zufällig erzeugten Testeingaben findet der gelenkte Fuzzer (Rot) den versteckten Fehler im Schnitt deutlich schneller (gestrichelte Linien). Er generiert die Eingaben nicht komplett neu, sondern mutiert und selektiert vorherige. So kann er evolutionär Testfälle erzeugen, die größere Teile der Schaltung aktivieren (höhere Coverage) und erhöht jeweils die Chance, den Fehler zu finden. Grafik: IMMS.

Während einer Hardware-Simulation wird das Verhalten der Schaltung bei einer bestimmten Eingabe analysiert, auf Fehlerzustände geprüft und anschließend ermittelt, welche Teile der Schaltung dabei aktiv waren. Das ist die Abdeckung, die diese Eingabe erreicht hat. Entsprechend einer Optimierungsstrategie wird nun eine Mutation ausgewählt und damit eine neue Eingabe erzeugt. Diese wird wieder in einer Hardwaresimulation der Schaltung übergeben, diese auf Fehlerzustände geprüft und die erreichte Abdeckung ermittelt. Um die nächste Eingabe zu generieren, werden die beiden zuvor erreichten Abdeckungen miteinander verglichen. Wurde die Abdeckung durch die letzte Mutation verringert, wird diese abgelehnt und eine neue Eingabe auf Basis der ursprünglichen Eingabe erzeugt. Konnte die Abdeckung durch die letzte Mutation erhöht werden, wird die neue Eingabe mutiert, um die nächste Generation der Eingabe zu erzeugen. Ob Mutationen angenommen werden oder nicht, wird durch die Optimierungsstrategie entschieden und kann z.B. bei einem heuristischen Approximationsverfahren wie dem Simulated-Annealing-Ansatz stochastisch erfolgen. Als Mutationen werden typische genetische Methoden eingesetzt, die sich insbesondere bei digitalen Eingaben gut veranschaulichen lassen. So kann z.B. die Eingabe 1011 durch den Austausch eines einzelnen Bits modifiziert werden (z.B. 0011) oder eine Position abgeschnitten (101), angehängt (10110) oder eingefügt (10101) werden. Derzeit unterstützt WTF die Simulationssoftware Cadence Xcelium und Icarus Verilog.

Der Funktionsumfang kann jedoch durch den modularen Aufbau des Fuzzers leicht erweitert werden. Insbesondere lassen sich neue Optimierungsstrategien integrieren.

Ausblick

Der Hardware-Fuzzer WTF befindet sich momentan noch in der Entwicklungs- und Erprobungsphase, hat aber bereits seine Funktionalität an Beispielschaltungen belegen können [3]. Der Vorteil vom „coverage-guided fuzzing“ zeigt sich gut im Vergleich zum „unguided fuzzing“, bei dem Testeingaben für die zu testende Schaltung komplett zufällig generiert werden (Abb. 2). In einem Versuch wurde WTF an einer Beispielschaltung getestet, in die am IMMS absichtlich ein Fehler eingebaut wurde. Dank des Feedbacks der durch die Testeingaben erreichten Abdeckung kann WTF effizient Testfälle mit höherer Abdeckung generieren und damit schneller den in der Schaltung versteckten Fehler finden.

Nachdem der Fuzzer die generelle Anwendbarkeit des Fuzzing-Konzepts im Bereich der Hardwareverifikation belegen konnte, wird er nun mit am IMMS entwickelten industriellen Chips getestet. Auch die von anderen Forschergruppen publizierten Ansätze werden sich nun dem Praxistest unterziehen müssen. Da sich „coverage-guided fuzzing“ im Softwarebereich erfolgreich als Verifikationsmethode etabliert hat, wird es spannend zu sehen, ob dies auch im Hardwarebereich gelingt.

Kontakt: Henning Siemen, M.Sc., henning.siemen@imms.de

Literatur:
[1] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, “Fuzzing hardware like software,” 2021. [Online]. Available: https://arxiv.org/abs/2102.02308
[2] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, “Rfuzz: Coverage-directed fuzz testing of rtl on fpgas,” in International Conference on Computer-Aided Design (ICCAD), 2018, pp. 1–8.
[3] H. Siemen, J. Lienke, and G. Gläser, “Hot Fuzz: Assisting verification by fuzz testing microelectronic hardware”, in 20th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), 2023

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Das Projekt KI-EDA wird vom Bundesministerium für Bildung und Forschung im Rahmen der Maßnahme „Mikroelektronik für Industrie 4.0 (ElektroniK I4.0)“ unter der Verbundnummer es2eli4001 gefördert, das IMMS unter dem Kennzeichen 16ME0010.

www.imms.de/ki-eda