Static Analysis for Proactive Security

Michael Huth[1(✉)] and Flemming Nielson[2]

[1] Department of Computing, Imperial College London, London SW7 2AZ, UK
m.huth@imperial.ac.uk
[2] Department of Mathematics and Computer Science,
Technical University of Denmark, 2800 Kongens Lyngby, Denmark
fnie@dtu.dk

Abstract. We reflect on current problems and practices in system security, distinguishing between *reactive* security – which deals with vulnerabilities as they are being exploited – and *proactive* security – which means to make vulnerabilities un-exploitable by removing them from a system entirely. Then we argue that static analysis is well poised to support approaches to proactive security, since it is sufficiently expressive to represent many vulnerabilities yet sufficiently efficient to detect vulnerabilities prior to system deployment. We further show that static analysis interacts well with both confidentiality and integrity aspects and discuss what security assurances it can attain. Next we argue that security models such as those for access control can also be statically analyzed to support proactive security of such models. Finally, we identify research problems in static analysis whose solutions would stand to improve the effectiveness and adoption of static analysis for proactive security in the practice of designing, implementing, and assuring future ICT systems.

1 Introduction

In the past 10–15 years, we witnessed a very substantial and increasingly accelerated transformation of Information and Communications Technology (ICT), the advent of smart phones and of digital social networks with global reach being two prominent examples. In addition, the emergent so called Internet of Things (IoT) and Cyber Physical Systems (CPS) are a recent but major development which could be highly disruptive in sectors not traditionally associated with ICT.

All of these systems or systems of systems contain software as crucial ingredients. The reliability of such software is traditionally assured through systematic testing as a best industrial practice. The limitations of this approach have been widely recognized, and its effectiveness has been somewhat improved through its combination with more formal techniques to validate critical software such as device drivers, and complemented with a range of other techniques such as manual code review. In fact, important formal methods such as type theories and static analyses are mature technologies that are routinely integrated in compilers and thus markedly improve the reliability of software – although programmers and IT project managers may be oblivious to that fact.

B. Steffen and G. Woeginger (Eds.): Computing and Software Science, LNCS 10000, pp. 374–392, 2019.
https://doi.org/10.1007/978-3-319-91908-9_19

Traditionally, commercial ICT software is deployed under a *Caveat Emptor* regime: producers of software tend not to accept any liability should the execution of their software cause any damage, even when run as intended. This regime was adopted for proprietary software but this ownership model was challenged by the open-source and freeware movements, which see software production and validation as a transparent, community-driven effort; the Linux operating system being a prominent and most impactful outcome of such efforts. Caveat Emptor is also not appropriate for software written for embedded systems, where software bugs may cause physical harm or loss of life.

But all software can contain errors, which may represent security vulnerabilities that could realize privileged access to systems, services or information. One may see such conceived wisdom as justification for the current liability regime and software validation practice. Yet, this is being challenged by the next wave of digitization that the IoT and CPS will bring about. For example, smart cars will become increasingly autonomous, so critical software components will have to meet very high correctness standards to ensure safety of passengers and those within the car's environment, and security vulnerabilities may be exploited to corrupt safety mechanisms. Therefore, security – to name Confidentiality, Integrity, and Availability – will no longer be an isolated concern but one that impinges on safety, reliability, and other concerns of future IoT/CPS systems. And liability models may shift: German law makers, e.g., are presently considering to make car manufacturers or even programmers liable for accidents caused by future, fully autonomous cars [7].

Standard engineering practice is tensioned by the increased blurring of system security and system safety aspects. To illustrate, we may not be able to apply redundancy and physical separation principles familiar from the aircraft industry in the domain of smart cars, where consumers expect to interface with their familiar devices such as smart phones and where cost and competitive pressure constrain engineering and validation. But we may realize *logical* separation, say, through security policies. The well known Jeep Cherokee hack [26], e.g., exploited a vulnerability in the car's entertainment system and the fact that the cellular provider did not restrict communication with that system to the car's internal systems, giving remote attackers' access to safety-critical components such as brakes. A security policy or modified default configuration that an attacker could not circumvent would have addressed this issue. These concerns extend to critical infrastructure such as electricity grids (e.g. the cyberattack on the Ukrainian power grid in 2015) and vital IT systems (e.g. the AnnaCry ransomware attack that severely disrupted some hospital services in the UK in 2017) – turning software reliability and resiliency into national security issues.

Security. The traditional understanding of security is that it is comprised of three components. *Confidentiality* is intended to protect the disclosure of data to third parties; it is intimately related to ensuring the *privacy* of citizens, and the protection of intellectual property. *Integrity* is intended to ensure the trustworthiness of data; it is intimately connected to ensuring the *authentication* of those who modify data and the control state of IT systems. *Availability* is intended to

ensure that systems remain operational even in the presence of an active adversary, e.g. in a denial of service attack. Much of the research in security focuses on achieving confidentiality and integrity while availability is substantially harder to attain due to the physical components that form part of the IT systems.

A security management in which software gets patched routinely, as in the so called *Patch Tuesday*, means owners of software are responsible for installing updates regularly. Such practices led to a *reactive* approach to security: a security problem in software gets discovered, a fix for the vulnerability is identified (if possible), and that fix is shipped as a software update to all systems that run that software. Moreover, this reactive approach is also used by the attacking ecosystems, where vulnerabilities are discovered and sold in a layered market of increasing capabilities: from a potential memory leak applicable to some systems to Ransom as a Service with complete attack capabilities, financial accounts, and so forth. This reactive approach is hardly satisfactory for security engineering, and will not do in future IoT and CPSs for reasons already alluded to.

Proactive Security. In contrast, *proactive security* is an approach to system security that uses a set of techniques to construct ICT systems or systems of systems that have almost no vulnerabilities (and thereby dramatically reduce the need for reactive security measures) and that incorporate exploit prevention or at least exploit mitigation into all phases of system construction – including design, implementation, and assurance activities.

Proactive security has been prominently advocated by Schell [50] in 2012. The need for it seems even more pressing now than when its first notable developments in system security were made. The initial efforts of developing proactive security facilitated the construction of IT systems living up to the demands of the famous "Orange Book", developed in the US with Schell as a leading contributor. This formed the basis for the current *Common Criteria* [1] standard that provides a systematic approach to the design and implementation of a variety of IT systems offering different levels of security guarantees. Schell laments that the use of proactive techniques seems to have given way to the use of more reactive techniques; in other words, that our current approach to system security takes a passive rather than an active, proactive approach.

There are several explanations for the current predominance of reactive security. Business pressures, such as time-to-market considerations, often demand the rapid construction and validation of products based on common product families or user feedback as seen in code production for social networks [13]. The construction of systems that meet the Common Criteria may be neither feasible nor appropriate in such use contexts. Another factor is perhaps that security engineering is rarely taught in the ICT and Computer Science curricula and that there is a global skills gap in cybersecurity professionals.

Formal Methods. The Common Criteria standard offers a range of *Evaluation Assurance Levels* that describe the amount of rigour exercised in the security validation. To meet the higher Evaluation Assurance Levels, it is not expected to formally validate the entire IT system in question, but it is emphasized that

its critical components must be validated through the rigorous use of formal methods. We can see this as a form of *risk management*: only higher such levels demand use of rigorous methods, and even for those higher levels is it too costly or presently not feasible to formally validate full functional and non-functional behaviour of complex IT systems – e.g. a code base of a few million lines of code.

But it is feasible to do such formal validation for critical components or code units – e.g. a discrete controller for a safety-critical component. Another good example of targeting a critical component with rigorous formal methods is the validation of the micro-kernel seL4 [31], which was fully formally verified; and one could then leverage the reliability and resiliency of that component to more complex systems that critically rely on it, e.g. a drone [27]. And formal methods are not confined to executable code: in [36], it is proposed to use generics and functional programming to get more trustworthy implementation types from UML models.

The Common Criteria don't endorse particular formal methods. This, too, is consistent with a risk-management approach to system assurance, which would seek to use a combination of techniques that best meets the requirements and constraints at hand. The formal methods used may range from validation carried out by semi-automatic proof assistants such as Isabelle (used to verify the micro-kernel seL4) and Coq (used in the verification/validation of a compiler [34]), to validation carried out by fully automatic model checkers and SMT solvers, and to validation using fully automatic and often very efficient static analysers.

Even if formal certification for Common Criteria is not sought, there is a strategic advantage in using proactive system security: in case a system built still contains exploits, the fact that is was built with that approach will enable a much better understanding of where those exploits may occur and what capabilities they may have, including what system components they may impact. In fact, we may see the emergence of new certification standards that reflect specific assurance needs of cyber physical systems and their application domains [5].

Static Analysis. Formal methods will continue to play an important role in future certification schemes. Here we focus on the many methods from *Static Analysis* or *Program Analysis* [43]. These techniques may use data-flow equations, constraint systems, abstract interpretation, type systems, and type and effect systems to mention just the most widely used ones here. A principal advantage that Static Analysis offers, above and beyond what other approaches in Formal Methods provide, is that its techniques typically realize analysis capabilities that come with low computational complexity – usually polynomial time, sometimes even linear time, rather than exponential time or worse. This makes them an attractive choice for proactive security engineering, certainly as a "first line of defence" that rules out certain security vulnerabilities at low computational cost and at almost no development or production cost.

Static analyses gain this advantage by making abstractions of the systems they analyze. This is typically an over-approximation of some precise analysis result – whose full precision is typically non-computable. Such abstraction and the compositional reasoning that this can support make Static Analysis a tool

set for ensuring the security of entire systems or systems of systems – which may span ICT, IoT, and CPS systems. Whilst it is a unique opportunity to explore the potential of Static Analysis in the next generation of digital systems, the challenge will be to engineer static analyses that make judicious trade-offs between their effectiveness (that the over-approximation still provides useful insights at the right level of abstraction, and with sufficient security assurances) and their cost (that the computation of results has sufficiently low complexity in terms of the scale of the system under analysis). Effectiveness here includes that analysis findings are reported in a manner that is useful for those who need to act on such results: ordinary programmers, modellers, and so forth. There is little research on this aspect, which is argued in [13] to be critical for transfer and adoption of static analysis in practice.

Overview. To meet this challenge, the development of security mechanisms for system engineering (e.g. security policies), the development of effective yet easily usable Formal Methods (in particular of static analyses) need to go hand in hand. In this paper, we will outline some key approaches, challenges, and further considerations that mean to provoke thinking and future research in this important area of security engineering for future digital systems. Admittedly, our exposition reflects a certain scientific bias, as it is not our intent to be encyclopedic in our treatment of static analysis and (proactive) security. Our express aim is, as already stated, to provoke thinking and to encourage new research in this important space whose future systems stand to profoundly impact us all.

2 The Security Landscape: Setting the Scene

As stated above, Static Analysis (or Program Analysis) [43] is mainly concerned with giving a sound over-approximation of the behaviour that a program may exhibit upon execution. If the sound over-approximation does not exhibit any malicious behaviour, this ensures that no malicious behaviour can arise during program execution. Security is largely concerned with ensuring that programs do not violate the confidentiality and integrity policies that are in place. Many of the key considerations of security have a strong analogy to questions studied in static analysis while some go a bit beyond. In this section, we will illustrate this close relationship because it is the basis for *why* static analysis forms a good foundation for ensuring proactive security – both for actual code and for the models that arise during software development.

Let us begin by explaining one of the fundamental static analyses traditionally used in compilers. *Definition-use* chaining aims at linking each definition of a variable (or assignment to a variable) to those uses of the variable where the value will be the one set at the definition (or assignment) point [43]. Soundness of definition-use chaining requires that we do not miss any uses; precision requires that we do not wildly over-approximate the set of uses.

To guard against errors in the formulation of the definition-use analysis one should prove that the analysis always soundly over-approximates. The first

problem to be addressed is the informality of such a formulation, one needs to be precise also for intricate features such as aliasing (where different variables are names for the same entity in storage). Many approaches can be explored to gain formality. Often, a good balance is found by using a so-called *instrumented semantics* [30], which explains actual code behaviour and keeps track of additional information, for example at which program point a variable was last defined. Then soundness of the definition-use analysis merely amounts to over-approximating the observations that can be made using the *instrumented semantics*.

For security, both confidentiality and integrity are guaranteed by assuring that information in ICT systems or socio-technical systems flows only in the intended and secure way. Control-flow integrity, e.g., ensures a program does not deviate from its normal control flow in order to initiate a privilege escalation attack. And the human decision of whether or not to open a certain web page should protect the confidentiality of personal information.

A key approach to security is through the study of information flow in programs as pioneered by [17]. Whenever we have an assignment, $x := \cdots y \cdots$, the value of y flows into x. We call this an *explicit flow* because the value of y is part of what is stored into x; we also call it a direct flow because it happens as the result of a single assignment. The analogy to definition-use chaining is immediate. Whenever we have an assignment $x := \cdots y \cdots$ at some program point, definition-use chaining would be able to tell us which of the program points defining y might influence the current definition of x.

Frequently, assignments are performed in bodies of conditionals, which thereby influence the decision to perform an assignment. As an example, for boolean variables x and y, there is hardly any difference to the behaviour of the program `if` y `then` $x :=$ `true else` $x :=$ `false` with respect to that of the program $x := y$. But the former has no explicit flow while the latter has.

The consideration of implicit flows takes care of this anomaly: there is an *implicit flow* from y to x whenever an assignment to x occurs inside the scope of a conditional that uses the variable y [17].

Apart from explicit and implicit flows, there are other and more suble forms of *covert flows* (paraphrasing the notion of covert channels). They may arise due to termination issues, timing issues, and dependencies between non-deterministic or parallel computations. However, we shall concentrate on the direct flows comprised of explicit and implicit flows as illustrated, and on the transitive closure of the direct flows – the latter traditionally referred to as *indirect flows*.

In our discussion of security concepts in Sect. 1, Confidentiality was explained as preventing disclosure of data to third parties; this amounts to ensuring the absence of indirect flow from the data to a use belonging to a non-trusted party. Similarly, Integrity was explained as ensuring the trustworthiness of data; this amounts to ensuring the absence of indirect flow to the data from a definition belonging to a non-trusted party. In summary, simple considerations of explicit and implicit indirect flows – which use modest generalisations of definition-use chains – suffice for ensuring Confidentiality and Integrity.

We are confident that approaches rooted in static analysis will continue to be useful when security policies grow in complexity as demonstrated in later sections. The main research challenge of static analysis is to ensure that the sound over-approximation is sufficiently informative (in excluding behaviour that cannot arise) while keeping the computational complexity at a manageable level (preferably close to linear).

The composition of systems considered to be secure (in isolation) does, too often, not result in a secure system. Running cryptographic security protocols "on top of each other" is a case in point. It remains a research challenge to facilitate the *compositional* construction of secure systems. While progress is being made, it is still beyond the state of the art to do so in general – let alone for IoT systems in which security, safety, and other concerns are co-dependent.

In the light of the lack of compositionality in security engineering, the low computational complexity of many static analyses may come to the rescue: It makes it feasible to perform whole-program analyses rather than attempting to achieve compositionality.

3 Static Analysis of Security Models

Static analysis is also applicable to models of IT Systems, not only source code or binaries. UML diagrams and access-control models are important examples thereof. Access-control models specify which subjects have access to what resources, and under which circumstances. Prominent examples are Role-Based Access Control (RBAC) [49], XACML (see e.g. [4]), and OAuth [2].

In RBAC, users are associated with one or more roles, and roles are associated with access permissions: a user gets a permission if she has a role with such a permission. This de-coupling facilitates scalability of specifications and change management of permissions. The core RBAC model has also been extended in numerous ways, for example with administrators who have permissions to make role-user assignments. XACML is a policy language in which one can specify the circumstances for granting access, based on attributes and their fine-grained combination. OAuth, on the other hand, is a protocol that is widely used on the web as it can give third-party applications limited access to an HTTP service, for example by giving the third party an access token as in the User Managed Access architecture of the Kantara initiative.

Instances of such access-control models specify the allowed and disallowed access within a system. Therefore, we need to validate that such instances capture intended access restrictions and permissions. Static analysis, and its close cousin *model checking* [9,42,51,53], can proactively validate such intent for instances of such access-control models. Extensions of RBAC, such as ARBAC that also provides support for administration, can be statically analyzed to determine whether models meet specified security requirements – for example that certain users or roles never gain certain access permissions (see e.g. [18,48]). The algorithms used may explore the state space exhaustively (provided sets of users, roles, and resources are finite) but are often too complex. While such

techniques may be seen to be static analyses, the static analysis tool box may be more fruitfully applied by devising provably sound abstractions of access-control models: for example, it may be possible to simulate role hierarchies through a temporal sequence of administrative actions, and so a security analysis may then be performed on a less complex simulation – an ARBAC system without role hierarchies and lower computational complexity.

The XACML models contain policies that consist of access-control rules as crucial ingredient. Validating an XACML model therefore benefits from statically proving that policies meet certain specifications. This is particularly important since languages such as XACML support access control in distributed, and potentially open systems. Therefore, we need security guarantees on the composition of policies and where a composition algebra may support a range of operators, e.g., logical ones such as Conjunction, and control structures such as Conditional Delegation. A prominent validation problem is to determine whether a policy has anything to say on an access request of interest; if not, this under-specification may be a potential vulnerability. Another validation problem is that the composition of policies may provide conflicting evidence for granting or denying an access. We also want support for reliable change management: if one policy is modified to another one, is the modified one a refinement of the original one in that it preserves important grant and deny decisions?

The work on PBel [11], was motivated by such questions and designed a rule-based policy-composition language in which basic rules where composed with operators expressible in Belnap's 4-valued logic (see e.g. [21]) and where these operators are functionally complete for that logic. Validation problems such as the ones discussed above, were then shown to be transformable into satisfiability problems over the predicates used in rules within policies. The approach made use of the 4-valued Belnap logic to capture not only grant and deny decisions, but also conflicts and under-specifications. That paper took an atomic view of predicates that build rules of PBel. But the semantics of PBel and its validation analyses would also work for richer predicates, for example those expressed in quantifier-free first-order logic. A nice example and application of how to interpret richer predicates for policy analysis in XACML is given in [47,55].

PBel was designed for studying the aforementioned problems, not for being used in practice. However, there is an opportunity in influencing the design of real-world access-control languages such that they support a formal and statically analyzable core, the full language is mere syntactic sugar of that PBel core, and the full language is user-facing. Such an approach is familiar from programming language design [52] and its benefits are clear: practical relevance since the full language is what users (here policy writers and administrator) want, support for proactive security through an analyzable core, and transfer of analysis from a core representation to a semantically equivalent full-language policy.

In fact, a core language may even be extended or equipped with interfaces to obtain a user-facing language that hides the concrete syntactic nature and semantics of the core. This may be particularly useful if such details are irrelevant or incomprehensible to those who specify and manage access control. To

illustrate, BelLog [54] is a datalog-like language for physical access control – extended to Belnap logic: in a building with a fixed topology of rooms and hallways, where each door has a digital lock, we seek simple policies for each lock that – in their entirety – enforce building-wide security policies such as "This room can only be accessed through previous entry into the lobby." In [54], it is shown how synthesis techniques for temporal logic can be adjusted to BelLog so that a solution to the synthesis problem realizes all specified security problems, and also maps this solution to local solutions for each digital lock. Moreover, local solutions are simple formulas of first-order logic that are easy to implement and enforce locally. One could imagine to extend this with synthesis techniques rooted in satisfiability of the temporal logic CTL, so that it becomes possible to specify and enforce security policies during the physical-layout design.

One challenge that we would then face, and that is often overlooked in academic research in static analysis, is that analysis results would have to be rendered in a form that is intelligible and actionable to the stakeholders of the application domain, in this example architects and physical security experts. As [13] pointed out, there is already need for more work on this when stakeholders are source code developers.

The case study in [28], e.g., considers this problem for a trust-aggregation language in which rules represent "trust signals" that are interpreted as real numbers. Such numbers are aggregated with composition operators, such as maximum or weighted average, to reflect how an overall computed score of all observed signals should support decision making. For example, whether or not to rent out a car to a client at a certain rate may be informed by a weighting of years of accident-free driving, the type of car, and so forth [29]. We then need to validate the manner in which such scores are aggregated, e.g., to rule out that this always supports the same decision. The tool developed in [28] reduced such validation analyses to satisfiability problems that an SMT solver could solve. But the reduction makes the evidence computed with such an automated theorem prover not meaningful to those who wrote the aggregation policy. Fortunately, it is possible to devise a static analysis over the semantics of the trust-aggregation language that renders an over-approximated but sound version of this evidence – and meaningful in terms of the aggregation semantics [28].

A good question in that context is what academics and practitioners can do to encourage a better alignment of foundational work and practical R&D in security engineering. One problem is that the value systems of academia and industry are not well aligned. For example, research on user-facing analysis reporting may find it hard to get into a top academic research conference. For another example, excellent foundational work may only be adopted in industry if funnelled through or integrated within industrial standards or if produced in-house.

4 Security Assurances: Information Leakage

Formal methods traditionally have promised to provide *absolute* guarantees of correctness – to the extent of providing a mathematical and flawless proof. However, it is easier to motivate the use of formal methods in software development

if it is presented as a way of enhancing the quality of software against errors and attacks. "Continuing the metaphor, we have found that software engineers more readily grasp the concept and practical value [...] if we dub it *exhaustively testable pseudo-code*." [41, p. 71] Moreover, it has been argued that methods which seek mathematical proof of program correctness can deliver such guarantees only for mathematical abstractions and not for programs as causal models within operational environments [16,19,40]. While this may suggest principal limitations of the reach of formal methods, the past decades have seen tremendous advances in foundations and applications of formal methods for software verification.

Clearly, formal methods operate on an *abstraction* of the real world system and it is a key lesson of security that abstractions pave the way for security holes. "Abstraction is an important concept we cannot do without when designing and understanding complex systems. [...] However, software security problems arise when intuitive properties of an abstraction do not match its concrete implementation." [23, p. 179] Indeed, even the hardware upon which software is executed is an abstraction. As an example, it is generally believed that computer memories will retain their values until explicitly changed or until power is cut off. However, cosmic radiation or even heat may make this abstraction invalid [25].

In the next three paragraphs we consider three key approaches to providing assurances of the correct use of static analysis for ensuring the security of systems. One of these takes its origin in traditional ways of ensuring the correctness of static analyses [43]. Another one goes back to techniques for establishing noninterference results that show the absence of information flow [56]; for example that no sensitive information is reaching unintended parties. The third approach replaces the qualitative view with a quantitative one by characterising the information leakage with respect to entropy [14]; this may support decisions of whether the computed leakage is acceptable or not.

Instrumented Semantics. Whenever we employ a static analysis we should establish its correctness – especially when safety, security, and their interplay is at stake. For some static analyses the notion of correctness is rather immediate, such as when we are analysing the values of some variables or perhaps the combinations of values of all variables. Assuming that we deal correctly with the bit strings that programs operate on – such as taking into account that integers have a maximal value and that the multiplication of two positive 32-bit integers is not necessarily positive – it is fairly obvious how to formalise correctness.

For other static analyses the notion of correctness is less immediate – this is typical of situations in which we analyse the past or future behaviour of programs. As a simple example, consider definition-use chaining where each definition (or assignment) of a variable is linked to all the potential uses of the value given to the variable at that point [43]. One way to formulate correctness is to consider potentially infinite execution traces. A more amenable way is to formulate an *instrumented semantics* that keeps track of certain elements of the manner in which computations are performed as well as the results they are intended to give. This suffices for proving the correctness of definition-use chaining as was discussed in Sect. 2.

More importantly, this set of techniques immediately generalises to handling the correctness of *explicit* information flows – both for confidentiality and integrity. These techniques can be augmented with considerations of the *implicit* information flows that occur due to conditional branching [44] as discussed in Sect. 2.

One advantage of the instrumented semantics approach is that the notion of correctness, once formalised, usually has a rather direct intuition (with respect to the overall security goals of the system), thereby reducing the risk for security holes due to abstraction. Also, it is feasible to extend the instrumented semantics approach with some of the more advanced security considerations such as declassification and endorsement where the security policy is deliberately violated at selected points [37].

An obvious disadvantage of the instrumented semantics approach is the possibility of basing correctness on an inadequate (read incorrect) instrumented semantics. Especially when dealing with non-determinism and parallelism it may be hard to correctly model the covert flows that arise.

Non-interference. The use of instrumented semantics is a qualitative approach requiring inspection of the way in which computations are performed and results produced. Another qualitative approach is that of non-interference, which only inspects results produced. Specifically, suppose an attacker may want to learn the values of some sensitive inputs to a program. The program satisfies non-interference if any variation in the input values of sensitive variables would not result in any observable difference in program outputs.

There are many different approaches to the formalisation of *non-interference* and we cannot touch upon all of them. In [22], it was required that observations on traces should be invariant under certain permutations of the actions in the traces. In [56], a simulation based approach was taken but only for deterministic and terminating programs. In [38], it was required that certain projections of traces should be equal; while there are clearly differences in the formal definitions, there also is a substantial amount of similarity, e.g. the trace based development of [38] reuses the proofs of the simulation based development of [56] (see [38, p. 15]). In [57], it is required that two executions should produce comparable sets of outcomes (thereby taking account of non-determinism) where non-termination is made observable (so as to avoid masking covert channels due to non-termination).

The main advantage of the non-interference approach is that we mitigate the risk of basing correctness on an inadequate instrumented semantics. A disadvantage of the non-interference approach is that it is still open to security holes due to abstraction since non-interference is usually established for models that are more abstract than a traditional instrumented semantics. More importantly, it is argued in [44, Sect. 8] that many formulations of non-interference fail to maintain a distinction between *confidentiality* and *integrity*, which constitute two of the key dimensions of the security landscape, and hence they fall short of convincing the security engineer of their relevance. (In short, non-interference is good at characterising the semantic influences but not whether they arise

due to confidentiality or integrity breaches.) Yet another disadvantage is that it may prove futile to establish a non-interference result for what would seem to be an acceptable security policy; this may arise because non-interference often is "asking for too much", and so in particular non-interference finds it hard to adequately incorporate cryptography as a way to achieve secure systems.

Regarding our discussion of the lack of compositionality in Sect. 2, it is fair to say that non-interference often just deals with the program in isolation whereas more complex considerations (such as non-deducibility on computation [33]) are required to regain some compositionality.

Entropy. For a more precise account of information leakage, one may consider quantitative approaches based on entropy. The basic assumption is that we have joint probability distributions available to characterise how sets of variables take their values. Given such data we can then define the amount of information that is derivable from an observation by its *entropy*. The assumption is that a program variable x is now a random variable taking values in a finite set V. *Shannon's Entropy* $H(x)$ is the expected value of information contained in each observation of x. This is an information-theoretic measure that is non-negative, additive for independent random variables, and monotone. Such intuitive properties characterize function H up to a constant. An important derived concept is that of *conditional entropy*: $H(x \mid y)$ denotes the portion of the entropy of variable x that is independent from another random variable y.

There are two extreme cases of the conditional entropy $H(x \mid y)$. One extreme case is where x, y are aliases for the same entity. Then, we will always make the same value observations for x and y: we obtain $H(x \mid y) = 0$ which indicates that we learn nothing further if we learn the value of x, given that we already know the value of y – the value of y determines the value of x. The other extreme case is where x and y are truly independent; then, we get $H(x \mid y) = H(x)$ – indicating that our previous knowledge of y tells us nothing of x.

The advantage of this quantitative approach to information flow is its ability to precisely quantify the amount of information $H(x) - H(x \mid y)$ that might be leaked due to information flow. This amount "should" be 0 if we have been able to prove a non-interference result, and if it is even slightly larger than 0 it "should" be impossible to establish a non-interference result. This suggests to accept a system as secure if the conditional entropy is sufficiently close to 0. A current disadvantage of this approach is that we have less tool support for analysing the security of systems according to information-theoretic, quantitative measures.

Perspective. In our view, non-interference is both demanding too much (in not permitting minute flows) and discriminating too little (in not distinguishing between confidentiality and integrity) to be useful for validating the use of static analysis in ensuring the security of systems. On the other hand, approaches based on instrumented semantics should interact well with state of the art in static analysis tools while methods based on entropy should be investigated as they offer to provide stronger assurances, and they can lead to metrics for the support of decision making in security engineering.

5 Discussion

A static analysis is subject to potentially conflicting aims. It needs to be *abstract* since many concrete properties of interest to it are undecidable for general programs and programming languages. Given that need for abstraction, it also needs to be *precise* enough, so that it will often enough arrive at findings that are digestible and useful to the analysts. But the static analysis should also be *sound*. By this we mean that the analysis models all possible executions of the program. This is important for security considerations: if some real executions are missed by the analysis (e.g. because the meaning of a language construct may depend on the implementation environment), these executions may be security vulnerabilities that an attacker might exploit. Finally, the static analysis should *scale* so that we can run it on large programs or code bases effectively.

The discussion around soundness superficially seems similar to a discourse documented partly in [16,19]: the question of whether formal verification can prove the correctness of an executing program with mathematical certainty. This was posed at a time when it was very difficult to promote use of formal methods into R&D and to get credibility for devising such methods. It is fair to say that we have come a long way! Major ICT companies such as Amazon, Facebook, and Microsoft are using a range of formal methods in tactical and strategic ways, and this was made possible by persistent research and tool building of formal-methods researchers in the past decades.

Formal methods such as static analysis are very useful for security engineering. First, if a static analysis fails to formally verify a property – such as the absence of memory leaks – at a higher level of abstraction, then this can also be a validation concern within the actual execution environment and so may require code modifications.

Second, static analysis tools may help us understand the scope of unsoundness that may occur when transferring reasoning that is sound at one abstraction layer to another one. For example, the work in [24,46] provides static analyses with which we may understand the differences and input sensitivities of programs between an idealized execution with mathematical real numbers and a finite-precision implementation of real numbers.

Third, code development operates at several abstraction layers and static analyses can certainly validate higher such layers in isolation. For example, the use of generics at the level of UML specifications [36] minimizes the risk of type incompatibilities in implementations; the use of static analysis to validate information flow of process models [6] is validating security properties of the process design itself; and the use of automated theorem proving in examining specified language standards such as those for Javascript [10] can flag up issues of interest to the standardization committees.

Fourth, there may be a compelling business case for formally verifying specific system components. For example, one may build an operating system in such a way that only its small micro kernel ever runs in supervisor mode – meaning that it runs at the most privileged level of the supporting hardware. There is then an incentive in formally verifying such a micro kernel if it is planned to be used in a

variety of security- or safety-critical systems. As already discussed, this has been done for the micro kernel seL4 [31]. A lot of the verification effort here went into assuring that the kernel will interact correctly with its environment, for example that the access control is correctly enforced, and that binaries of the kernel correctly implement the C semantics of its source code. Change management is a challenge for such efforts, and the authors discuss in [31] the degree of severity with which different types of kernel code changes impact the overall verification effort. The DARPA initiative [27] demonstrated that use of such a formally verified micro kernel can significantly harden the security and resiliency of systems that rely on it, for example a drone that white-hack teams can no longer compromise.

The ability to deal with change is one of the main selling points of any security validation method for software and executing systems in practice. The paradigm shift from single to multi-core CPUs, and even the advent of GPU and FPGA development environments provide exiting research opportunities for language design, compiler technology, and static analysis as tools for producing secure software on heterogenous or bespoke hardware. But they also challenge conceived ideals and models of computation, such as memory consistency and thread schedulings and force us to rethink the use of static analyses in this setting. Technological innovations in isolation technology, such as Intel's SGX [15], also mean that static analyses for proactive security may have to be adjusted to reflect such innovations and their isolation architectures.

Another important trend we see is the recognition that verification tools and static analyzers should be the object of verification themselves. While this invites an infinite logical regress, it makes perfect sense from an engineering perspective. For example, Cadar and Donaldson predict in [12] that, by 2025, the analysis of static analyzers and entire compilers will be common place. There are already efforts at producing compilers that are provably correct within the abstraction level of such reasoning; let us mention the CompCert project [34] that offers a mathematical proof that the compiler introduces no bug in the convertion from source code to binary, and the CakeML project which work on verifying system implementations of substantial parts of Standard ML (see e.g. [32]). And there is already some work on certifying the results of model checkers [39]. These efforts are related to the need to better understand how static analyzes can be adapted to best support code development within professional development environments. We refer to [35] for a discussion of such needs.

We think that static analysis and its practical use can be furthered by use of big data and data analytics. Static analysis and formal verification of a software system will no doubt make that system more secure. But reactive security mechanisms may be needed to improve the resiliency of that system at run-time; for example, to prevent RowHammer attacks that aim to compromise security by breaking an abstraction [8]. It then makes sense to base such a reaction on available data for security vulnerabilities, the probability of them turning into active security threats, and the system impact that exploits – which realize such threats – may have. In fact, one may use formal techniques such as solvers or

optimizers to reason about how best to devise such reactive security postures and their evolution [20,45].

A static analysis tool may also use data and quantitative analytics to determine a "scheduling" of which bugs to report and why. For example, this may inform the ordering or prioritizing of such reporting. Past exploit data, the potential impact path of a program point or stochastic assumptions about program input may inform which bugs to report. Put in another way, if a security engineer has time to look at 4 bugs, which ones should the static analysis present to her? Such rankings may even include prioritizations based on risk appetite or specific attack models. Indeed, we see such work already happening in the realm of security operations centres (SOCs) and the use of data mining and artificial intelligence in enterprise platforms (see e.g. [3]), where one concern is to understand human behaviour and where "bugs" are now potentially suspicious human behaviours that may be worth reporting: which ones to report may well depend on a particular concern an analyst has.

6 Conclusion

For a long time, IT systems have been central to our society but they are becoming increasingly complex, pervasive, and autonomous. This development offers many benefits to society but also creates risks related to the safety and security of societies relying on the correct functioning of their IT systems. Furthermore, globalisation and the Internet of Everything mean that software is becoming a commodity for which a system integrator may have little insight in the way its code has been developed nor how it performs in corner cases that trade off soundness and completeness of a static analysis.

Computer Science not only offers the software and algorithms making this development of today's IT systems possible – it also provides key methods and techniques for ensuring the correct behaviour of complex and inhomogeneous IT systems. Compositionality lies at the heart of a component-based approach to the construction of IT systems. Formal methods may be used to harden critical components. But too frequently the composition of secure components does not result in a secure system (as was discussed for cryptographic protocols). Also, compositional security seems an even harder goal when it comes to the IoT systems of systems that will shape our future in the Internet of Everything.

Static analysis is noteworthy among the formal methods approaches in offering a variety of analyses of low computational complexity that therefore are likely to scale to systems built out of many components. We have argued that a number of security considerations related to confidentiality, privacy, integrity and authenticity can be addressed using techniques from static analysis. These techniques apply equally well to existing code, to access control policies, and to designs (or models) of systems under development. Further advances in static analysis are likely to go beyond the mere optimization of software in order to fully tackle the challenge of proactively ensuring the security of complex IT/IoT systems.

Acknowledgements. We expressly thank Marieke Huisman, Alan Mycroft, and David Schmidt for their very useful comments on drafts of this paper. The first author was supported in part by the UK EPSRC, through the grants EP/N023242/1, EP/N02334X/1, and EP/N020030/1, and by funding from Intel Corporation. The second author was supported in part by the IDEA4CPS Research Centre studying the Foundations for Cyber Physical Systems funded by the Danish Research Foundation for Basic Research (DNRF86-10).

References

1. Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org
2. OAuth 2.0. IETF OAuth WG. https://oauth.net/2/
3. Status today: Artificial intelligence that understands human behavior. https://www.statustoday.com/
4. eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard, 22 January 2013. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html
5. Framework for cyber-physical systems. Release 1.0, May 2016. US NIST, Cyber Physical Systems Public Working Group
6. Accorsi, R., Wonnemann, C.: Static information flow analysis of workflow models. In: INFORMATIK 2010 - Business Process and Service Science - Proceedings of ISSS and BPSC, 27 September–1 October 2010 in Leipzig, Germany, pp. 194–205 (2010)
7. Adee, S.: Germany to create world's first high-way code for driverless cars. Online title, 21 September 2016. New Scientist
8. Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., Oren, Y., Austin, T.M.: ANVIL: software-based protection against next-generation rowhammer attacks. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, 2–6 April 2016, pp. 743–755 (2016)
9. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
10. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised JavaSript specification. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 87–100 (2014)
11. Bruns, G., Huth, M.: Access control via Belnap logic: intuitive, expressive, and analyzable policy composition. ACM Trans. Inf. Syst. Secur. **14**(1), 9:1–9:27 (2011)
12. Cadar, C., Donaldson, A.F.: Analysing the program analyser. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, 14–22 May 2016 - Companion Volume, pp. 765–768 (2016)
13. Calcagno, C., et al.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_1
14. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. J. Comput. Secur. **15**(3), 321–371 (2007)
15. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptology ePrint Archive 2016:86 (2016)

16. DeMillo, R.A., Lipton, R.J., Perlis, A.J.: Social processes and proofs of theorems and programs. Commun. ACM **22**(5), 271–280 (1979)
17. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
18. Ferrara, A.L., Madhusudan, P., Nguyen, T.L., Parlato, G.: VAC - verifier of administrative role-based access control policies. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 184–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_12
19. Fetzer, J.H.: Program verification: the very idea. Commun. ACM **31**(9), 1048–1063 (1988)
20. Fielder, A., Panaousis, E.A., Malacaria, P., Hankin, C., Smeraldi, F.: Decision support approaches for cyber security investment. Decis. Support Syst. **86**, 13–23 (2016)
21. Fitting, M.: Bilattices and the theory of truth. J. Philos. Logic **18**(3), 225–256 (1989)
22. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 26–28 April 1982, pp. 11–20 (1982)
23. Gollmann, D.: Computer Security, 3rd edn. Wiley, Hoboken (2011)
24. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_17
25. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: 2003 IEEE Symposium on Security and Privacy (S&P 2003), Berkeley, CA, USA, 11–14 May 2003, pp. 154–165 (2003)
26. Greenberg, A.: The Jeep Hackers are Back to Prove Car Hacking Can Get Much Worse, 8 January 2018
27. Hartnett, K.: Computer Scientists Close In on Perfect, Hack-proof Code, 23 September 2016
28. Huth, M., Kuo, J.: Quantitative threat analysis via a logical service. Technical report 2014/10, ISSN 1469–4174, Department of Computing, Imperial College London (2014)
29. Huth, M., Kuo, J.H.-P.: On designing usable policy languages for declarative trust aggregation. In: Tryfonas, T., Askoxylakis, I. (eds.) HAS 2014. LNCS, vol. 8533, pp. 45–56. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07620-1_5
30. Jones, N.D., Nielson, F.: Abstract interpretation: a semantics-based tool for program analysis. In: Handbook of Logic in Computer Science, vol. 4, pp. 527–636. Oxford University Press (1995)
31. Klein, G., Andronick, J., Elphinstone, K., Murray, T.C., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2:1–2:70 (2014)
32. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 179–192 (2014)
33. Lanotte, R., Maggiolo-Schettini, A., Troina, A.: Time and probability-based information flow analysis. IEEE Trans. Softw. Eng. **36**(5), 719–734 (2010)
34. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)

35. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.-Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. Commun. ACM **58**(2), 44–46 (2015)
36. Murphy, R.: Increasing assurance levels through early verification with type safety. J. Cyber Secur. Inf. Syst. **3**(2) (2015). https://www.csiac.org/journal-article/increasing-assurance-levels-through-early-verification-with-type-safety/
37. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, 5–8 October 1997, pp. 129–142 (1997)
38. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing robust declassification and qualified robustness. J. Comput. Secur. **14**(2), 157–196 (2006)
39. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_2
40. Nelson, D.A.: Deductive program verification (a practitioner's commentary). Mind. Mach. **2**(3), 283–307 (1992)
41. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
42. Nielson, F., Nielson, H.R.: Model checking *Is* static analysis of modal logic. In: Ong, L. (ed.) FoSSaCS 2010. LNCS, vol. 6014, pp. 191–205. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12032-9_14
43. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-662-03811-6
44. Nielson, H.R., Nielson, F.: Content dependent information flow control. J. Logical Algebraic Methods Program. (2016, in press)
45. Livshits, B., Katz, O.: Toward an evidence-based design for reactive security policies and mechanisms. Technical report, November 2016
46. Putot, S.: Analyse statique de programmes et systèmes numériques. Tech. Sci. Inform. **33**(1–2), 159–162 (2014)
47. Kencana Ramli, C.D.P., Nielson, H.R., Nielson, F.: The logic of XACML. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 205–222. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35743-5_13
48. Ranise, S., Truong, A., Armando, A.: Boosting model checking to analyse large ARBAC policies. In: Jøsang, A., Samarati, P., Petrocchi, M. (eds.) STM 2012. LNCS, vol. 7783, pp. 273–288. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38004-4_18
49. Samarati, P., de Vimercati, S.C.: Access control: policies, models, and mechanisms. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 137–196. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45608-2_3
50. Schell, R.R.: Current cybersecurity best practices - a clear and present danger to privacy. Keynote, ERCIM News 90 (2012). http://ercim-news.ercim.eu/en90/keynote
51. Schmidt, D., Steffen, B.: Program analysis *as* model checking of abstract interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49727-7_22
52. Schmidt, D.A.: The Structure of Typed Programming Languages. Foundations of Computing Series. MIT Press, Cambridge (1994)
53. Steffen, B.: Data flow analysis as model checking. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 346–364. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54415-1_54

54. Tsankov, P.: Access control with formal security guarantees. Ph.D. thesis, Computer Science, ETH Zurich (2016)
55. Turkmen, F., den Hartog, J., Ranise, S., Zannone, N.: Analysis of XACML policies with SMT. In: Focardi, R., Myers, A. (eds.) POST 2015. LNCS, vol. 9036, pp. 115–134. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46666-7_7
56. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996)
57. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: 16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), Pacific Grove, CA, USA, 30 June–2 July 2003, p. 29 (2003)