Subproject B4: Verifying Software and Reconfigurable Hardware Services

Eric Bodden[1], Marie-Christine Jakobs[3], Felix Pauck[1], Marco Platzner[1], Philipp Schubert[1], Heike Wehrheim[2]

1 Department of Computer Science, Paderborn University, Paderborn, Germany
2 Department of Computer Science, Oldenburg University, Oldenburg, Germany
3 Department of Computer Science, TU Darmstadt, Darmstadt, Germany

1 Introduction

Subproject B4 focuses on designing quality assurance measures for single services that are (i) purchased, (ii) composed into service compositions, and (iii) directly employed at runtime in an on-the-fly manner. These measures must allow one to check if an acquired IT service actually fulfills the properties as promised by the service provider. The techniques for ensuring the quality of services must further enable their users to quickly check whether the desired properties hold without forcing them to expensively analyze the service and verify the properties themselves. In this subproject, we consider service providers that assemble compositions and compute centers that execute services as users. The target of the quality assurance measures are individual IT services that are offered in an OTF market. Since services might be implemented in software or synthesized in reconfigurable hardware components, measures to check both must be created.

For example, a service composition used for image recognition may rely on a software or hardware service implementing a filter that is used as an image preprocessor. Hence, it must be ensured that this preprocessing service is safe to use. To this regard, *safety* stands for the property that no error location can be reached.

To reach these goals, Subproject B4 has proposed *proof-carrying services*. Proof-carrying services come with a proof in form of a certificate that allows its users to efficiently check whether the certificate and therefore the properties that the service claims to hold are valid or not, instead of requiring them to extensively analyze and compute the proof for the target service. The idea is to shift the computational expense of verifying the desired properties of an IT service to its respective provider. With respect to software services, the technique implemented for creating and checking certificates is called *proof-carrying code (PCC)* or, in case of reconfigurable hardware components, *proof-carrying hardware*

eric.bodden@uni-paderborn.de (Eric Bodden), jakobs@cs.tu-darmstadt.de (Marie-Christine Jakobs), felix.pauck@uni-paderborn.de (Felix Pauck), platzner@uni-paderborn.de (Marco Platzner), philipp.schubert@uni-paderborn.de (Philipp Schubert), heike.wehrheim@uni-oldenburg.de (Heike Wehrheim)

(PCH). Besides PCC, the *Programs-from-Proofs (PfP)* technique has been proposed, which follows the same goal. However, in contrast to PCC, PfP does not attach certificates or proofs to a service, but instead uses the proof to transform the program (service) into an equivalent program for which the properties of interest can be verified more easily—the service provider thus still verifies the original program whereas the user only has to verify the transformed program. PfP is in depth described as one of the subproject's selected topics in Section 2.1.

In the area of PCH, we have proposed techniques to verify functional and non-functional properties. As an example, in Section 2.2 we elaborate on certifying memory access monitors for reconfigurable hardware systems. In such systems, different modules need to access shared memory, and predefined static or dynamic memory access patterns describe legal access sequences. A memory access monitor is a runtime module that captures these patterns and blocks illegal accesses. Certifying such monitors instead of the complete modules greatly reduces the required computational effort.

Under the term *hardware/software-co-verification (HW/SW-co-verification)* Subproject B4 has developed techniques that pair PCC and PCH. These techniques target services or programs that use so-called *custom instructions* to trigger reconfigurable hardware components. In order to pair PCC with PCH, pre- and postconditions are computed during software verification, such that the hardware verification must assure that these conditions hold. These conditions become part of the certificate and, hence, must only be computed by the service provider, which further unburdens the user. The selected topic presented in Section 2.3 provides more information about HW/SW-co-verification.

In both areas, software (PCC) and hardware (PCH), only safety properties were initially considered. Later on in the project, the focus shifted to the more challenging—with respect to verification/analysis complexity—security properties. This shift required the design of novel techniques as well as the implementation of new frameworks and tools. In Section 2.4, we present the novel PHASAR framework that we developed as part of Subproject B4. PHASAR allows one to statically analyze software written in languages from the C family. We use it to design and prototype new analysis algorithms and strategies to effectively compute safety *and* security properties (and their proofs) for the target services.

Existing mature static analysis tools were also used to create certificates for security properties. These tools usually provide no proof; hence, the quality of the certificate relies on the quality or accuracy of the analysis. Therefore, instruments to determine the accuracy of analyses become indispensable. Consequently, in Section 2.5 we take a closer look at benchmarking software analyses.

2 Selected Research Topics

2.1 Programs-from-Proofs

The goal of Subproject B4 is to provide approaches that let consumers (users) of software or hardware services efficiently and automatically check whether a service ensures the desired properties. One means to achieve this goal is to apply the principle of proof-carrying code (PCC). To achieve efficient checking, PCC relocates the major workload of

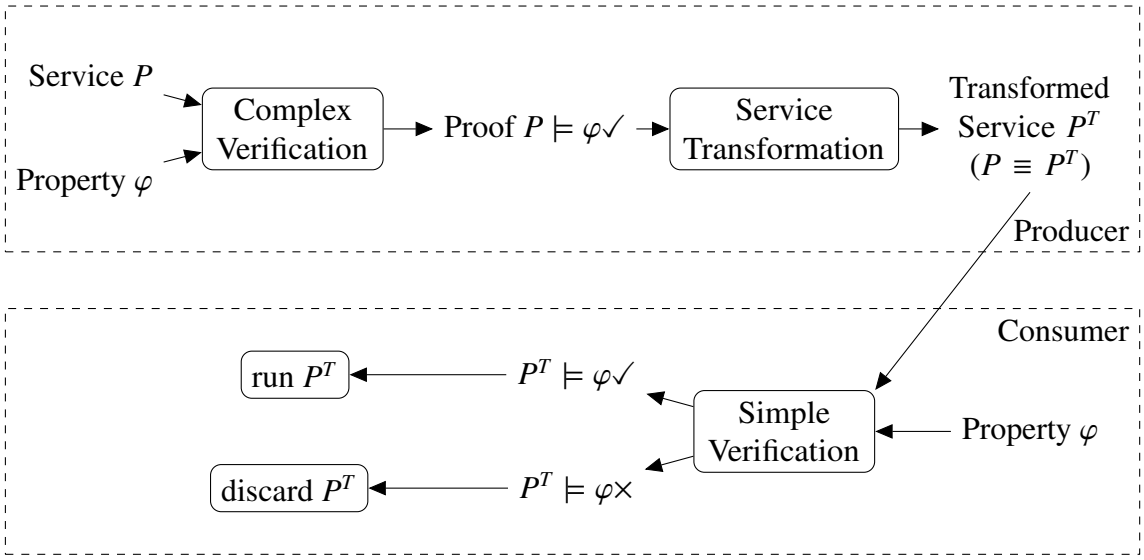

Figure 33: *Generic workflow of the Programs-from-Proofs approach.*

checking: namely, performing the proof that the service ensures the desired properties, to the service producer. The producer then attaches the generated proof in form of a certificate to the service. Hence, the consumer only needs to check whether the certificate attests that the service ensures the desired properties, which is typically assumed to be more efficient than proof generation. Several PCC instances for various properties and analysis techniques have been suggested, some relying on mathematical proofs and others on more general concepts of proofs, such as abstract state spaces. However, often these approaches suffer from large certificates. Furthermore, consumers are bound to specific validation approaches often tailored to the type of certificate and cannot apply existing verification technology. In addition, proof generation is not automatic for all PCC instances.

To overcome these issues, Subproject B4 has proposed an alternative principle, named Programs-from-Proofs (PfP). Like PCC, PfP is a generic principle that still forces the service producer to perform the work-intensive part of proof generation. However, it goes without certificates and lets the consumer employ existing, but relatively efficient verification techniques, such as dataflow analyses instead of specific validation techniques, for example. To achieve this, PfP uses the insights that the structure of a software service (i.e., program) can heavily influence the complexity of verification but that many proofs, in particular proofs that model the (abstract) state space of a (software) service, restructure the service such that its verification becomes simpler. More concretely, PfP employs the proof to transform the (software) service into a different, but behaviorally equivalent and property preserving (software) service that is easier to verify.

Figure 33 shows the generic workflow of the PfP approach, which we explain in more detail in the following.

1. Initially, the producer verifies (software) service P with respect to property φ, applying a potentially complex and costly verification. During the complex verification, many PfP instances use a combination of a computational expensive, incremental analysis and a cheap analysis. The cheap analysis is responsible for checking the property while the main purpose of the expensive analysis is to restructure the (abstract) state space, i.e., to restructure the paths of the analyzed (software) services by

unfolding loops, avoiding reintegration of branches, excluding infeasible execution paths, etc., such that the cheap analysis succeeds in property checking. To achieve the necessary restructuring, the expensive analysis often performs the restructuring incrementally based on the failed proof attempts of the cheap analysis.

2. After the producer's verification attempt succeedes, the producer uses the proof to automatically transfer the restructuring that was done for proving the property to the service. Thereby, it is important that the transformation (1) does not change the service's (functional) behavior, (2) keeps the validity of the property, and (3) ensures simple verification of the property on the transformed (software) service P^T. All PfP instances Subproject B4 has developed focus on proofs in form of abstract reachability graphs (ARGs). An ARG is a representation of the abstract state space of a (software) service. Important for the PfP instances is that all ARG paths correspond to syntactic paths in the analyzed (software) service and that all syntactic paths that are also semantically feasible (i.e., the executable paths) are represented in the ARG. However, an ARG and the (software) service likely structure paths differently and the ARG may contain less infeasible syntactic paths. All those differences allowed the cheap analysis component to prove the validity of the property. The ARG characteristics mentioned above are the reasons why the PfP instances, which Subproject B4 has developed, all translate the ARG, in particular its paths with their structure, into a (software) service, which becomes the transformed service P^T delivered to the consumer. Furthermore, these characteristics allow one to verify the desired behavioral equivalence of the (software) service before and after transformation.

3. Once the consumer has received the transformed service P^T, he or she performs a simple verification of the transformed service P^T to efficiently and automatically check whether a service ensures the desired property φ. If the complex verification consisted of a combination of cheap and expensive analysis as described above, the simple verification typically applies (a variant of) the cheap analysis technique, although the consumer might use a different implementation of the cheap analysis. Our PfP instances even allow the cheap analysis to become path-insensitive. Typically, our instances each use a variant of the respective cheap analysis that performs an efficient, flow-sensitive dataflow analysis. The reason is that any path sensitivity that the cheap analysis contributed during complex verification is also incorporated in the ARG structure and, thus, in the transformed (software) service. To prevent the consumer from harm, the simple verification must be tamper-proof, i.e., it must detect any tampering of the process that invalidates the desired property on the received service, e.g., deviations in properties, invalid producer proofs, incorrect transformation, or changes to the transformed service during delivery. Hence, the simple verification must be sound, i.e., it must ensure that only services that fulfill the desired property are verified successfully. Since soundness is typically guaranteed by the simple verification technique itself, we focused on showing successful consumer verification in a tamper-free PfP workflow. More concretely, for the PfP instances Subproject B4 has developed, we have proven that the simple verification will succeed if the complex and simple verification consider the same property, the complex verification has succeeded, and the simple verification verifies the services computed by the transformation based on the proof generated by the complex verification.

4. Depending on the outcome of the simple verification, the consumer lastly either runs the transformed service in case of a successful verification or otherwise discards the service.

Our proof-of-concept instance for PfP [WSW13] has addressed typestate properties, protocol-like properties enhancing types with information about their state, and has introduced the idea to transform ARGs into services (programs). Its complex verification combines predicate model checking and a typestate analysis, while the simple verification performs a pure typestate dataflow analysis. A typestate analysis allows to decide whether certain operations are possible with respect to the typestate of a variable. For example, an integer variable may be in the typestate uninitialized, demanding that it is initialized before it is used. Subsequent PfP instances [JW15; JW17] have extended the supported types of analyses and properties, but reuse the idea of ARG to service (program) transformation. Furthermore, we have used the software analysis framework CPAchecker [BK11], a tool that supports configurable program analysis, to implement our PfP instances. While we have reused CPAchecker's existing analyses and its possibility to combine analyses to realize the complex and simple verification, we have integrated the ARG to service (program) transformation into CPAchecker. Practical evaluations of our PfP instances with CPAchecker have shown that the consumer's simple verification is indeed significantly more efficient in terms of runtime and memory usage than the producer's complex verification. Also, PfP is often more efficient than existing PCC approaches applicable to configurable program analyses.

The PfP approach here makes a first essential contribution in the range of the proof procedures. As described before, PfP addresses the problem that proofs stored in the proof-carrying code method are usually very large and therefore inefficient to handle. It could be shown that this can succeed by means of PfP to embed the proof quasi partially directly into the structure of the program which can be analyzed. Thereby, the size of the proof is reduced and nevertheless the possibility of the efficient proof examination by the user remains. As a result, PfP thus allows for an often more efficient examination of the necessary evidence and a more efficient transfer of this evidence to the user. However, another advantage of PfP over PCC is also the reduction in *trusted base*: In PCC, the user must trust the verification procedure, which itself is often relatively complex (albeit runtime efficient). In PfP, however, this checking procedure corresponds to a relatively simple data flow analysis, which should increase confidence that this procedure is error-free. PfP thus increases confidence in the overall security of the corresponding services.

2.2 Proof-Carrying Hardware

Proof-carrying hardware (PCH) was first proposed by Drzevitzky et al. [DKP09; DKP10] as the reconfigurable hardware equivalent of PCC. The PCH concept distinguishes a circuit producer (e.g., a design center) and a consumer, e.g., a data center operating a reconfigurable computer or an embedded system based on a reconfigurable system-on-chip. The consumer loads and executes reconfigurable hardware modules that were created by the producer. Additionally, the consumer specifies a security property that the modules need to fulfill and, before loading, requires formal proofs of the properties. It is the task of the producer to generate not only the modules but also the proofs and transmit both to

the consumer. The consumer will verify that the proofs are correct and actually belong to the modules. In PCH, the compound of module implementation and the proof have been denoted as a *proof-carrying bitstream*.

An important security property for reconfigurable hardware systems pertains to memory access policies. The density of today's reconfigurable hardware devices allows for implementing reconfigurable systems with a large number of modules or cores, respectively. Through dynamic or even partial reconfiguration, modules can be loaded on demand, increasing flexibility. Several modules that access the same physical memory need to adhere to a specified policy governing their access patterns. A simple static policy, for example, is to enforce that each core can only access its own segment of the memory. There are, however, more involved policies in use when it comes to intended sharing of data between cores, the handling of conflict-of-interest classes, or different security levels. Huffmire et al. [HSKL08] introduced a *monitoring-based approach* to ensure memory access security. They presented a formal language and a compilation tool flow that allows a designer to specify a memory access policy and generate a circuit for a so-called memory access monitor. All modules' memory accesses have to be routed through the monitor, enabling the monitor to block any memory access that violates the policy at runtime.

We guarantee memory access security in the strength of formal verification by bringing together the monitoring approach of Huffmire et al. with the proof-carrying hardware concept. The consumer operates a reconfigurable resource where several cores access shared memory and memory accesses are routed through a memory access monitor that implements a predefined memory access policy. The policy can change during runtime to reflect different applications and security requirements. The consumer receives a new monitor together with a proof of its functional correctness, verifies the proof and, in case of success, partially reconfigures the monitor.

Our tool flow starts with the consumer that uses behavioral Verilog to specify the memory access policy. The producer receives the design specification and synthesizes it into an FPGA bitstream, using the tools of Huffmire et al. and, subsequently, VTR for Verilog synthesis and place & route. After that, the producer re-extracts the logic function from the bitstream and, together with the original design specification, computes the miter function. The miter function is shown in Figure 34 and is constructed such that the output of the miter, i.e., the error flag, can only be 1 if the specification and implementation differ for at least one input vector. For proving functional equivalence for combinational circuits, it is thus sufficient to prove unsatisfiability of the miter. We use ABC to construct a miter in conjunctive normal form and the SAT solver PicoSAT to prove unsatisfiability. PicoSAT also generates a proof trace that, together with the bitstream, forms the proof-carrying bitstream.

Dynamic memory access policies lead to sequential monitor circuits. Thus, we extended the concept and tool flow to also work with sequential miters using bounded sequential equivalence checking [WDP14]. A sequential miter circuit is unrolled for a specified number n of time frames, resulting in n copies of the circuit that are connected at their flip flops. Every time frame represents one clock cycle, and we can change the primary inputs and observe the primary outputs at every individual cycle. The miter construction then compares all outputs in each time frame and the flip flop signals of the last frame, and raises the error flag if there is a deviation somewhere. As we have to choose a specific amount of unrolling time frames, we observe that the compiled monitors are essentially

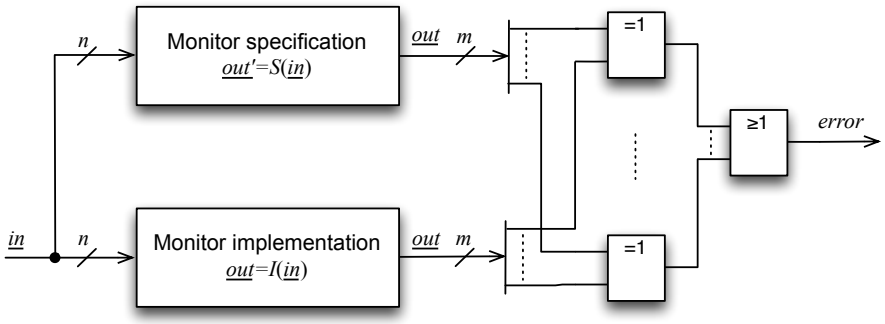

Figure 34: *Miter $M(S(x), I(x))$ for proving the functional equivalence of specification S and implementation I, (taken from [WDP14]).*

state machines, and their internal transitions only depend on their current state and the new input. Suppose there is an input sequence i that satisfies the miter function, i.e., it leads to different outputs for the implemented circuit and the specification, and the corresponding state transition path of the state machine contains cycles. Then the input sequence i', which leaves out all state cycles of i, is also a valid input sequence and it also satisfies the miter. If the miter is thus provably unsatisfiable for all maximum length-cycle free state transition paths, it is unsatisfiable for all input sequences of all lengths. Hence, we can simply use a number n of unrolling frames larger than the number s of automaton states to ensure that every cycle-free sequence has been considered.

The consumer receives the bitstream for the monitor circuit together with the proof trace for unsatisfiability. In a first step, the consumer also extracts the monitor's logic function from the bitstream and forms a miter in conjunctive normal form in the same way as the producer, but with the original specification. The so-created miter is compared to the miter sent by the producer, which is part of the proof trace. If the miters do not match, then the proof is not based on the desired functionality and the monitor is refused. If the miters match, the consumer verifies the proof by checking each reduction step in the proof trace until an empty clause results. Only then, is the implementation shown to adhere to the security property and the monitor accepted.

To demonstrate the capability of our proposed approach for ensuring memory security, we built a prototypical system. As platform we chose a ZedBoard containing a Xilinx Zynq-7000 system-on-a-chip with a dual ARM Cortex-A9, and 512 MB RAM. Our prototype architecture embeds a *virtual FPGA overlay* into a reconfigurable system as shown in Figure 35. We use a virtual FPGA since we need to be able to interpret the transmitted configuration bitstream for the memory access monitor. FPGA vendors typically do not share the necessary information, and reverse engineering the bitstream or additionally transmitting and interpreting low-level circuit descriptions such as Xilinx XDL are extremely tedious processes. Virtual FPGAs or FPGA overlay architectures have become increasingly popular in the last years for a number of reasons. They provide a means to implement portable circuits, bring partial reconfiguration capabilities to FPGAs that have no native support of this feature, achieve fast configuration rates, prototype coarse grained arrays, or be able to implement circuits created with open source tool flows such as VTR on real FPGAs.

We leverage the virtual FPGA overlay ZUMA and embed it into ReconOS [LP09]. Re-

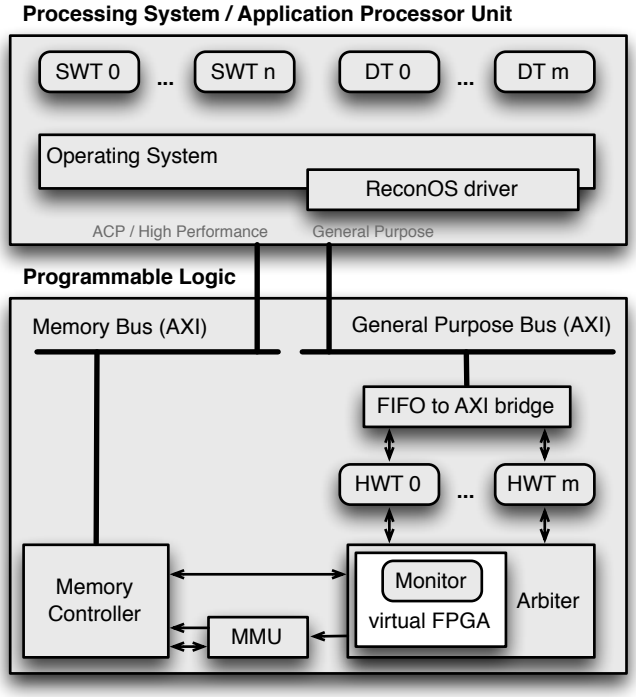

Figure 35: *Xilinx Zynq version of ReconOS, with $n + 1$ software threads (SWT), $m + 1$ hardware threads (HWT), their $m + 1$ delegate threads (DT), and an arbiter including a memory monitor in the memory access path of the HWTs (taken from [WDP14]).*

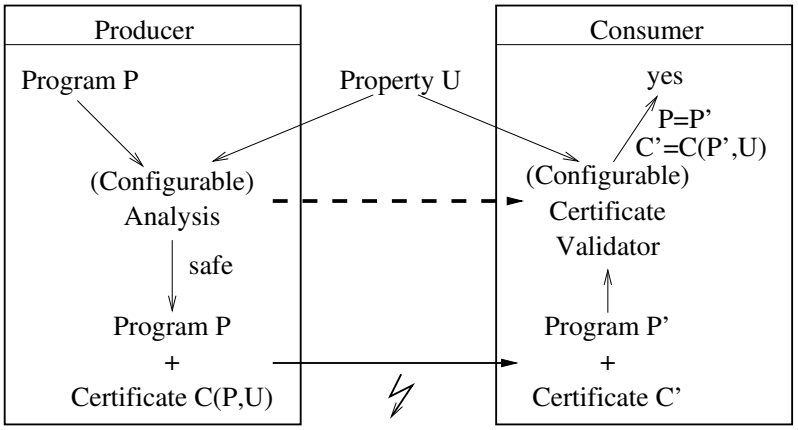

Figure 36: *Certification generation and validation.*

conOS is an execution environment for hybrid hardware/software systems featuring a multithreaded programming model that allows for regular software threads as well as hardware threads. The use of ReconOS enables us to use a mature, Linux-based infrastructure for implementing hardware/software systems, including a CPU core, memory controller, peripherals and a standard software operating system. As shown in Figure 35, we have modified the ReconOS arbiter in the memory access path of the hardware threads to include the memory access monitor. The access monitor itself is implemented in our ZUMA virtual FPGA overlay. For the inputs, the arbiter provides the monitor the virtual memory address, the type of the request (read or write) and its source, the hardware thread identifier.

We further presented a series of experiments to investigate different aspects of our approach and prototype. The experiments showed that the approach is feasible and can secure static and dynamic memory access policies of different complexities. With 61.84% to 90.53% of the overall workload, depending on the memory access policy, the producer clearly bears the computational burden of establishing the consumer's trust in the module. As expected, the overlay comes with rather high area and delay overheads. The reduction of these overheads was also addressed.

2.3 Proof-Carrying Code and Its Relation to Proof-Carrying Hardware

The core principle underlying the work of Subproject B4 was to enable on-the-fly checking of service correctness by attaching proofs as witnesses to the correctness of both software and hardware. Here, we briefly explain our technique of proof-carrying code (software verification) and its integration with hardware verification.

For proof-carrying code we employ analysis and verification techniques that can formally prove the validity of properties in software programs. Hence, we can employ the proofs as a form of *certificate* to a service's correctness. The basic principle of proof-carrying code is the idea that the generation of certificates (on the side of the service producer) can be time-consuming while its validation (on the side of the consumer) should be easy. Figure 36 depicts this basic scheme. The producer develops a program (service) P, which

should adhere to property (requirement) U. The producer is supposed to carry out the costly analysis (proving the holding of property U on P). The outcome of the analysis, more specifically the correctness proof, is then attached to the program in the form of a certificate. When a consumer wants to use this service, it retrieves program and certificate from some repository. Our assumption here is, however, that neither producer nor storage in repositories can be fully trusted. Thus, the consumer might actually receive a slightly different program P' or a slightly modified certificate C'. Our technique enables the consumer to quickly validate whether the certificate still fits to the program and thereby whether the received program P' meets the intended requirement U.

Instead of developing a certification technique per property or per class of properties, we have investigated the generation of certificates for arbitrary properties [JW14] via a *configurable certification process*. Our generic approach builds on an existing framework for configurable program analysis with tool support in the form of CPAchecker [BK11]. CPAchecker executes an analysis *meta algorithm* generating a (structured) abstract reachability set of a given program. The meta algorithm can be steered by a number of user-supplied inputs (e.g., telling CPAchecker when to stop the analysis and when to merge states). This presents a way of uniting different program analysis techniques, ranging from data-flow analyses, to computing abstract information for control flow graphs, to model checking, computing a tree-like abstract structure. The generated reach set is then subject to property checking.

For the certification process, we use the—anyway generated—reach set as certificate. Similar to the analysis, we develop a generic configurable certificate validation framework with a corresponding meta algorithm for certificate checking. In addition, we provide a way of (in a large number of cases automatically) generating the configuration of the certificate validation from a given configuration of the analysis. Our approach is tamper-proof in that the certificate validator only outputs "yes" if the program P remains unchanged ($P = P'$) and the obtained (and possibly corrupted) certificate C' is a valid certificate for the program P with respect to a desired property U. We have implemented our technique within the CPAchecker framework, and evaluated it on a number of different analysis techniques. For all of these, certificate validation is faster than analysis. We proved soundness of all of our techniques, i.e., we have shown them to be tamper-free.

To connect to the certification on the hardware level, we have studied how software certificates relate to the underlying hardware used for execution. Software analyses typically rely on the correctness of the processor hardware executing the program. More specifically, the strongest postcondition computation used to determine the successor state of a given state for a program statement assumes that the processor correctly implements the statement's semantics. Certificate validation heavily employs the strongest postcondition computations. This assumption of correct hardware is certainly valid for standard processors, since they undergo extensive simulation, testing, and partly also formal verification processes. However, during the last years processors with so-called custom instruction (CI) set extensions became popular, which challenge this correctness assumption. Customized instructions map a part of an application's data flow graph to specialized functional units in the processor pipeline in order to improve performance and/or energy efficiency.

In [JPWW14], we have presented a novel formal approach for software/hardware co-verification, in particular for processors with custom instruction set extensions. It (partially) employs the certificate computed by the software analysis to derive requirements on the

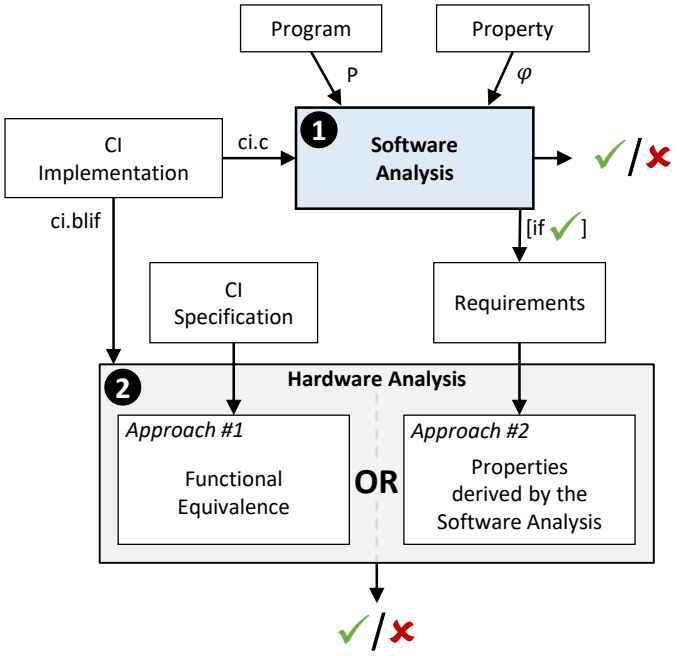

Figure 37: *Overview of hardware-software co-certification.*

hardware. These requirements then need to be validated in order for the software analysis to produce trustworthy results. Figure 37 gives an overview of our approach.

We have studied two different approaches for integrating software and hardware analyses that differ in what needs to be verified on the hardware side. Our first approach proves functional equivalence between the specification and the implementation of a custom instruction, e.g., that an integer adder is actually adding integer values. While proving equivalence is potentially the most runtime-consuming approach, it is also the most powerful, as it inherently covers all behavioral properties of the custom instruction on which software analyses could rely. Our second approach ties together software and hardware analyses more closely by exploiting the abstract state space of the program generated during verification to identify the specific properties of the individual program statements the software analysis has actually used during verification. These properties become *requirements* on the hardware. We thereby tailor the hardware verification exactly to the needs of the software analysis, hoping to avoid unnecessarily complex and runtime-consuming hardware verification.

We have built a toolchain automating all the steps of our approach, which are (1) the software analysis computing requirements on the hardware via the use of a verification tool plus information about the custom instructions, (2) the hardware analysis synthesizing property checkers from requirement and custom instruction specification, and (3) a SAT solver for checking satisfiability of the custom instruction implementation together with the property checker. We have evaluated our technique on different custom instructions occurring in programs using several software analyses for requirements extraction. As a main result from our experimentation, we can conclude that while tailoring the hardware verification more to the concrete needs of the software analysis indeed generally results in lower computational effort, neither approach is superior for all cases.

2.4 Static Analysis with PHASAR

Another selected topic of Subproject B4 is embodied in the genesis and development of the meanwhile well-known PHASAR [SHB19] project for static program analysis. PHASAR is a modular static analysis framework targeting the C and C++ programming languages and has been built on top of LLVMto account for the lack of general infrastructure for the analysis of such programs. PHASAR's infrastructure allows one to quickly draft prototypes for new program analyzers, novel algorithms and analysis strategies, and also allows for their evaluation.

We have built several novel analysis approaches on top of PHASAR, which we present in the following paragraphs.

C/C++ languages are often used for projects that require a direct interface with operating systems or hardware components. They offer control to programmers for creating efficient programs, but also require correct usage to avoid bugs or security issues. Compilers such as GCC and Clang and additional tools, such as Cppcheck and Clang Static Analyzer, aid in creating secure software. However, they often provide only simple checks or have a large number of false or missed warnings due to imprecise analysis. For Java programs, program-analysis frameworks such as Soot, WALA, and Doop provide more precise dataflow analysis. This type of implementation was not available for C/C++. This is where PHASAR came in, a novel program-analysis framework designed for LLVM infrastructure. It can be used for dataflow problems, call-graph construction, and points-to information. PHASAR is intended for static analysis and complements LLVM toolchain features. Some parts may be used as a compiler pass.

C/C++ programs can represent an entire software product line using static conditionals called features. Traditional static analysis techniques cannot be applied to software product lines directly, because the process of generating and analyzing all software products becomes prohibitively expensive due to the possibly exponential number of software products. To solve this problem, VARALYZER, a family-based approach was developed, which analyzes a software product line as a whole. VARALYZER transforms preprocessor directives into ordinary C code using a configuration-aware type checker. It supports not just analyses encoded in IFDS but also those encoded in interprocedural distributive environments (IDE). VARALYZER outputs the fully context- and flow-sensitive dataflow facts along with a feature constraint describing the product configurations for which they hold. This allows developers to find bugs and vulnerabilities much earlier in the development process, when a preprocessor has not yet even been applied, for instance, in a version-control system. The effectiveness of VARALYZER has been evaluated using a typestate analysis that checks for the correct usages of OpenSSL's Envelope (EVP) APIs on 95 compilation units. Challenges related to evaluating VARALYZER on full SPLs are detailed in [SGP+22].

MODALYZER [SHB21] is a novel approach that enables the scaling of static analyses on large software projects. The approach involves the pre-computation of summaries for parts of code that do not frequently change, which can be integrated into larger analysis scopes. The summaries can be seen as proofs of the property the client analysis attempts to demonstrate. Whole-program analysis (WPA), which can be memory-intensive and cause runtime problems, can be substituted with intra-procedural analyses that are simple enough to scale, as demonstrated by tools such as Clang-tidy and Cppcheck. However, semantic

program analyses such as shape, typestate, and dataflow analyses require detailed program representations that include the effects of procedure calls, which are impossible to scale if calculated for the entire program.

ModAlyzer provides a compositional approach to program analysis that is capable of scaling static context-sensitive, field-sensitive, and flow-sensitive inter-procedural program analysis. This is achieved through the compositional computation of analysis information. The success of the compositional analysis depends on the number of reusable parts of the application, for example, libraries, or parts that do not change from one analysis run to the next. Black Duck's recent study shows that 96% of the applications they scan contain open-source components, and those components now account for, on average, 57% of the code. The application of compositional analysis can accelerate the analysis of applications by reusing analysis results from previous runs, especially as open-source dependencies are updated much less frequently than application code.

Although previous work on compositional program analysis has been limited to certain types of dataflow analysis, ModAlyzer provides a mechanism for analysis dependency management for a fully compositional analysis that automates updates whenever new information becomes available that affects existing information. The approach also involves an efficient summary format that is able to persist general data. ModAlyzer can potentially scale the analysis of applications by reusing analysis results from previous runs.

Last but not least, IncAlyzer was developed to support summarization and reuse of static analysis information for *frequently* changing parts of a program. It assumes that the target project is developed using a version control system and aims at maximizing the reuse of static analysis information computed on a previous revision of the target project that is still valid. Summarization techniques can be used to pre-compute summaries that can be reused while analyzing the actual application code and may decrease the analysis time by a large factor. Tree-adjoining languages and Dyck context-free language reachability can help to increase the number of useful summaries. Incremental analysis can improve scalability for frequently changing code, as changes made to a program are usually small and thus should only cause invalidation of a small amount of the analysis results. Existing incremental static analysis techniques ignore the information provided by version control systems (VCS) and are only concerned with dataflow information.

Contrary to the Reviser approach, which only considers the dataflow parts of a client analysis for its incremental analysis and computes the code delta based on the inter-procedural control-flow graphs, IncAlyzer makes the complete client analysis stack (control-flow, callgraph, points-to, type-hierarchy and dataflow information) incremental and uses VCS information to obtain the code delta directly. If IncAlyzer recognizes that a code change has no impact on the semantics of the program while producing commit-annotated IR, no reanalysis is performed on the IR. IncAlyzer has great potential to allow developers to check-in persisted static analysis results directly to the VCS managed code repository for each commit of a project which are then both kept in sync throughout the continuous integration development of the project. This has the advantage that each revision only needs to be analyzed once. Any developer can check out a code revision accompanied by its respective up-to-date analysis results, allowing them to check and reuse them for incremental analysis locally. This allows static analysis information for each commit to be viewed as "certificate" which can be checked instantaneously for each given commit, according to the precision and capabilities of the underlying client analysis, of course. One

may even bind those "certificates" to the code, e.g., using cryptographic hashing, to avoid accidental or intentional manipulation.

2.5 Benchmarking with ReproDroid

The number of research communities fostering open science is steadily increasing. For instance, the software engineering community has turned artifact evaluations from a rarity into a standard. Funding agencies nowadays join this effort by rewarding the availability of open science artifacts. For these reasons, instruments to drive reproducible evaluations have become more important and needed than ever before. Building such instruments, in particular in the context of on-the-fly computing, proves to be challenging, since the market and its ecosystem must be available and accessible. With ReproDroid [PBW18], a framework that allows to create or adapt benchmarks so that these can be executed and evaluated automatically, we have proposed such an evaluation instrument for Android taint analysis. We have used ReproDroid to evaluate whether six "Android taint analysis tools keep their promises" [PBW18], to create, execute and evaluate a real-world benchmark [LPP+22] and to evaluate cooperative analyses [PW19].

Android *taint analyses* track the flow of sensitive data throughout one or multiple apps. Whenever sensitive information is accessed via a private *source*, it is marked as tainted and tracked through the app's data (and control) flow. If tainted data reaches a public *sink*, a data leak is reported in form of a *taint flow* that stretches from source to sink. We differentiate *intra-app* taint flows inside a single app from *inter-app* taint flows between apps.

To evaluate taint analysis tools, benchmarks are usually employed. A *benchmark*, in this context, consists of two parts: a set of apps and its *ground truth*, which is a complete list of all taint flows occurring in these apps. Since it is often difficult to determine whether a ground truth is correct or complete, micro benchmarks are often used. *Micro benchmarks* consist of tiny apps that were only implemented for benchmarking purposes. Each micro benchmark app usually implements only a single taint flow that uses or exploits a specific Android or programming language feature. Hence, the ground truth can be defined by documenting this specific taint flow only.

In the past, a benchmark's ground truth was often described in natural language, which allowed different interpretations and ultimately led to irreproducible results. ReproDroid uses the Android app analysis query language (AQL) [PBW18; PW19] to precisely specify a benchmark's ground truth and to interact with arbitrary Android taint analysis tools. Figure 38 provides an overview of ReproDroid's toolchain. First, the benchmark refinement and execution wizard (BREW) takes a set of apps as input. During Step 1, the sources and sinks that occur in these apps are identified. BREW allows to automate this process by automatically selecting sources and sinks which are specified in a configurable list. Such lists are typically used by taint analysis tools to identify the respective statements. Furthermore, for each pair of source and sink that belongs to the same benchmark case, it is specified whether it describes an expected or a not-expected taint flow. While an *expected* taint flow should be found by an analysis, a *not-expected* taint flow should explicitly not be found—finding it is considered to be a false positive result. Once the ground truth is fully described in BREW, the benchmark is ready to be executed. To do so, BREW

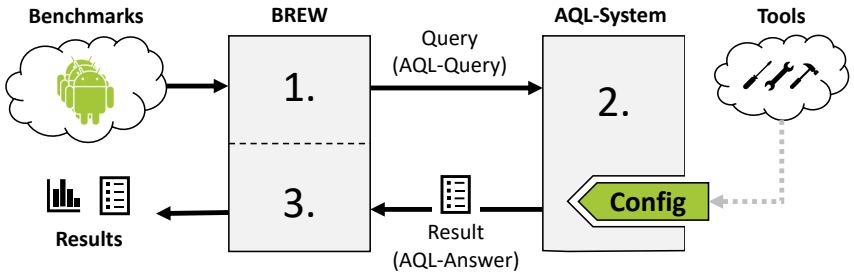

Figure 38: *Sketch of the* ReproDroid *toolchain.*

forwards one AQL query per benchmark case to the next component: namely, the AQL System, which performs Step 2 (see Figure 38). As the name suggests, the AQL System is the default system for using the AQL. In case of a query asking for taint flows, the AQL System looks up a taint analysis tool in its configuration and runs it in order to answer the query. If required, the taint analysis tool's output is converted into the AQL answer format. This answer is replied to BREW, which then compares the answer against the ground truth to finally compute the accuracy metrics precision, recall and F-measure (Step 3). These metrics summarize the benchmark's outcome and allow us to compare the performance of different tools.

In a first study [PBW18], we have used ReproDroid to check the feature and accuracy promises given for six taint analysis tools. For example, it is claimed that FlowDroid, the most-cited tool, is context-, flow-, field-, object-sensitive and lifecycle-aware and that it achieves certain precision, recall and F-measure scores for the DroidBench benchmark. Additionally, it is claimed that these tools are able to analyze real-world apps—a promise that we have also attempted to validate. In conclusion, we have found that most promises were kept by most tools. However, all of them seemed to struggle in case of real-world scenarios.

Initially, we have used ReproDroid to adapt the most-used (with respect to citations) micro benchmarks for Android taint analyses (DroidBench and ICC-Bench) such that they can automatically be executed and evaluated to guarantee reproducibility and comparability. Later, the real-world benchmark TaintBench [LPP+22] was created with and for ReproDroid. TaintBench comprises 39 malware apps that have been shipped via various app markets. For these 39 apps, 203 expected and 46 not-expected taint flows have been determined manually and specified in ReproDroid. Even though this ground truth is most likely incomplete, through the definition of expected and not-expected taint flows we are still able to evaluate taint analysis tools on this baseline. In the end, TaintBench has allowed us to gain novel and measurable insights that reveal capabilities and inabilities of analyses especially while handling real-world scenarios. Most surprisingly, it has also allowed us to detect regressions between two versions of two state-of-the-art analysis tools (Amandroid and FlowDroid) that were not visible using micro benchmarks only.

Combinations of analyses (*cooperative analyses*) can also be evaluated by means of ReproDroid [PW19]. In this case, the AQL is not only used to interact with arbitrary analysis tools but also to steer the cooperation between analysis tools, e.g., how to combine their results. To efficiently execute cooperative analyses, the AQL System allows to distribute the execution of different tools onto distinct and distributed AQL Systems. We have composed four cooperative strategies that overall employed 12 analysis tools in order

to deal with four analysis challenges. One of these strategies, for instance, deals with inter-app communication. This strategy allows to detect taint flows that start in one app and end in another. To do so, a taint analysis tool is queried to find intra-app taint flows and a combination of two additional tools to find inter-app flows. By means of the AQL, these intra- and inter-app flows are stitched together which has ultimately allowed us to detect taint flows across app boundaries. In case of all four challenges (reflection, native code, inter-component, and inter-app communication) significant improvements were able to be achieved through cooperation and measured via ReproDroid.

In the context of on-the-fly computing, cooperative analyses can be interpreted as service compositions themselves, i.e., each analysis tool represents a service, an AQL query describes the service composition, and the AQL System stands for a service provider, whereas another AQL System may take the role of a compute center. In this scope, ReproDroid can be used to determine the quality of services and service compositions. Due to the reproducible nature of benchmarks executed via ReproDroid anyone (consumer or producer) is able to check whether certain properties (e.g., accuracy metrics) are accomplished by a service (composition). Trustworthy and demonstrably accurate (cooperative) analyses can then be used for the "certification" of other services or service compositions.

3 Impact and Outlook

Subproject B4 has worked on various proof-carrying service techniques throughout all three periods of the CRC 901. In the beginning, the fundamentals of proof-carrying code (PCC) and proof-carrying hardware (PCH) have been examined closely, extended, and implemented in first prototypes. The evaluations conducted along the way have already proven the potential of these techniques in the context of on-the-fly computing, i.e., safety properties of services, to be used in service compositions, they could be certified by service providers (producers), and they were less expensively checked by their users (consumers, e.g., compute centers). Next, mainly during the second period, (1) PCC and PCH have been brought together such that software and hardware services interacting with each other can be certified collectively, (2) mature implementations have been developed to extend the field of application, such that more versatile (software and hardware) services and properties, in particular security-related properties, can be analyzed and certified, and (3) techniques to assess the quality of analyses have been researched and implemented. For example, the Phasar analysis and the ReproDroid benchmarking framework were constructed. While approaching the end of the CRC, the benefits of the effort spent so far became measurable not only in terms of more than 100 publications contributed by Subproject B4 but also in terms of available and usable artifacts, which have and will cause impact beyond research.[13] In the following, we present and discuss these benefits in the context of the five selected topics detailed above.

Based on the core scheme of proof-carrying code, we have investigated a number of optimizations and extensions (e.g., Programs-from-Proofs). The goal, for instance, was to provide more compact certificates. We have furthermore studied certification techniques for *hyperproperties*, more specifically for information flow properties [TW18]. Information flow analysis investigates the flow of data in applications, checking in particular for flows

[13] `https://ris.uni-paderborn.de/project/12` (19.04.2023)

from private sources to public sinks. Flow- and path-sensitive analyses are, however, often too costly to be performed every time a security-critical application is run. We have proposed a variant of proof-carrying code for information flow security. To this end, we have developed information flow certificates that get attached to programs as well as a method for information flow certificate validation. The technique has also been implemented within the program analysis tool CPAchecker [BK11]. Furthermore, we have studied different security policies for information flow and their integration in a certification context [TW18].

Programs-from-Proofs (PfP) represents one of these proof-carrying code (PCC) optimizations for which we have in turn proposed several extensions. The first PfP extension supports reachability properties and any kind of dataflow analysis as cheap analysis. Hence, complex verification becomes a combination of predicate model checking and an arbitrary dataflow analysis, named predicated dataflow analysis, while the simple analysis uses the dataflow analysis alone. Later, a generic PfP framework [JW17] has added support for arbitrary properties expressible as property automaton (including typestate and reachability properties). In addition, the framework allows to combine arbitrary expensive and cheap analyses in the complex verification as long as the cheap analysis solely checks the property, it is at least flow-sensitive, and both analyses are expressible in the framework of configurable program analysis [BK11], which allows to describe arbitrary abstract-interpretation based analyses. The simple analysis then uses the cheap analysis reconfigured as a dataflow analysis. Not only the generic PfP framework but all our PfP instances rely on the existing concept of configurable program analysis to describe the analyses: in particular, complex and simple verification as well as the combination of expensive and cheap analyses. As a last extension we have also adopted the idea of PfP to perform runtime verification with no overhead. These extensions and in particular the generic framework, show that the Programs-from-Proofs technique is highly applicable with respect to various properties and services. In conclusion, due to our research and implementations, the PfP approach has become a usable approach instead of a mostly theoretical concept.

Besides developing proof-carrying hardware (PCH) frameworks for certifying functional equivalence for combinational and sequential circuits, we presented PCH approaches for certifying non-functional security properties such as the worst-case execution time of hardware modules and keeping predefined error bounds for approximated circuits. For the demonstration of the PCH concept, we relied first on abstract FPGAs that could only be simulated, and later on virtual FPGA overlays that allowed us to show the feasibility of PCH on real FPGA hardware. In more recent work, we studied PCH as a tool for detecting hardware trojans in reconfigurable modules and showcased these methods on Lattice FPGAs with their known bitstream formats. Lastly, we want to mention that the proof-carrying hardware term that was introduced in the context of this subproject has been taken up by others [LJM12]. This silently demonstrates the impact of our research conducted in this area.

The concepts of PCC (software) and PCH (hardware) are built on the notion of a *certificate* certifying the correctness of software or hardware with respect to specified requirements. For the software, this is (in our project) a compact version of the abstract reachability graph (ARG) constructed during software verification. On the consumer side, the ARG is checked for two properties: (1) its fit to the program, i.e., whether it is an abstract reachability graph for the program, and (2) its consistency with the requirement, i.e., whether it actually proves

program correctness. While we used these certificates to realize the collaborative analysis of software and hardware services, such certificates have also recently been employed in software verification competitions such as SV-COMP [Bey22]. SV-COMP is an annual competition for software verification mainly targeting C programs. The tools participating in the competition have to determine whether a specified requirement is met or not. In the first case, tools are required to provide *correctness witnesses*, in the latter, *violation witnesses*. The correctness witnesses serve the same purpose as our certificates (and almost take the same form). Witnesses are then also checked for their soundness using so-called witness validators. This usage of certificates in competitions indicates and exemplifies that the concepts also proposed by Subproject B4 are adopted and used by others.

We started the PHASAR project in 2016 and made the first version publicly available in 2018 in a full-day workshop at the *39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI) conference. As of today, the PHASAR github repository has achieved 773 stars, was forked 123 times, and has grown far beyond the scope of Subproject B4 as 41 developers from around the globe contributed to the framework.[14] Moreover, each of the mentioned extension of PHASAR (MODALYZER, INCALYZER, VARALYZER—see Section 2.4) is accompanied by a research paper that includes extensive evaluations of the respective approach. Each paper, in turn, comes with an evaluated artifact that provides the option to reproduce the presented results. In summary, PHASAR has become a mature analysis framework that is evaluated, recognized and adopted by research and industry.

With REPRODROID we contributed an open source framework that allows anyone to evaluate analyses automatically and in a reproducible fashion on given benchmarks. Consequently, REPRODROID simplifies the benchmarking process, which was often performed manually before. Therefore and since evaluations such as benchmark executions are indispensable to show the effectiveness and efficiency of analyses, REPRODROID was not only used by us in our five subsequent publications to drive the associated evaluations but also by others. This versatile usage of REPRODROID best shows its impact in the community. In future, it could even become more important as a driver for competitions in the area of Android taint analysis, for example. Please note that each of our publications involving REPRODROID comes with an evaluated artifact and/or an open source repository. The related repositories in sum acquired 65 stars. All frameworks, tools and benchmarks released are also available on the respective website of the CRC.[15]

In summary, Subproject B4 has left its mark in the area of proof-carrying services or, in general, on soft- and hardware verification and analysis. Due to the publications made as well as the implementations and artifacts contributed, this mark has become persistent such that future researchers and practitioners can take our ideas, understand our results, use our tools and frameworks, and continue what has been started.

Bibliography

[Bey22] BEYER, D.: Progress on Software Verification: SV-COMP 2022. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS*

[14] Github repository: `https://github.com/secure-software-engineering/phasar` (04/24/2023)
[15] `https://sfb901.uni-paderborn.de/projects/tools-and-demonstration-systems`

2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Ed. by Fisman, D.; Rosu, G. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 375–402.

[BK11] Beyer, D.; Keremoglu, M. E.: CPAchecker: A Tool for Configurable Software Verification. In: *CAV*. Ed. by Gopalakrishnan, G.; Qadeer, S. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190.

[DKP09] Drzevitzky, S.; Kastens, U.; Platzner, M.: Proof-carrying Hardware: Towards Runtime Verification of Reconfigurable Modules. In: *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2009

[DKP10] Drzevitzky, S.; Kastens, U.; Platzner, M.: Proof-Carrying Hardware: Concept and Prototype Tool Flow for Online Verification. In: *International Journal of Reconfigurable Computing* 2010 (2010)

[HSKL08] Huffmire, T.; Sherwood, T.; Kastner, R.; Levin, T.: Enforcing memory policy specifications in reconfigurable hardware. In: *Computers & Security* 27 (2008), no. 5–6, pp. 197–215.

[JPWW14] Jakobs, M.; Platzner, M.; Wehrheim, H.; Wiersema, T.: Integrating Software and Hardware Verification. In: *IFM*. Ed. by Albert, E.; Sekerinski, E. Vol. 8739. Lecture Notes in Computer Science. Springer, 2014, pp. 307–322.

[JW14] Jakobs, M.; Wehrheim, H.: Certification for configurable program analysis. In: *SPIN*. Ed. by Rungta, N.; Tkachuk, O. ACM, 2014, pp. 30–39.

[JW15] Jakobs, M.; Wehrheim, H.: Programs from proofs of predicated dataflow analyses. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*. Ed. by Wainwright, R. L.; Corchado, J. M.; Bechini, A.; Hong, J. ACM, 2015, pp. 1729–1736.

[JW17] Jakobs, M.; Wehrheim, H.: Programs from Proofs: A Framework for the Safe Execution of Untrusted Software. In: *ACM Trans. Program. Lang. Syst.* 39 (2017), no. 2, 7:1–7:56.

[LJM12] Love, E.; Jin, Y.; Makris, Y.: Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition. In: *IEEE Transactions on Information Forensics and Security* 7 (1 Feb. 2012), no. 1, pp. 25–40

[LP09] Lübbers, E.; Platzner, M.: ReconOS: Multithreaded Programming for Reconfigurable Computers. In: *ACM Transactions on Embedded Computing Systems (TECS)* 9 (1 Oct. 2009), 8:1–8:33

[LPP+22] Luo, L.; Pauck, F.; Piskachev, G.; Benz, M.; Pashchenko, I.; Mory, M.; Bodden, E.; Hermann, B.; Massacci, F.: TaintBench: Automatic real-world malware benchmarking of Android taint analyses. In: *Empir. Softw. Eng.* 27 (2022), no. 1, p. 16.

[PBW18] Pauck, F.; Bodden, E.; Wehrheim, H.: Do Android taint analysis tools keep their promises? In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. Ed. by Leavens, G. T.; Garcia, A.; Pasareanu, C. S. ACM, 2018, pp. 331–341.

[PW19] Pauck, F.; Wehrheim, H.: Together strong: cooperative Android app analysis. In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Dumas, M.; Pfahl, D.; Apel, S.; Russo, A. ACM, 2019, pp. 374–384.

[SGP+22] Schubert, P. D.; Gazzillo, P.; Patterson, Z.; Braha, J.; Schiebel, F.; Hermann, B.; Wei, S.; Bodden, E.: Static data-flow analysis for software product lines in C. In: *Automated Software Engineering* 29 (Mar. 2022), no. 1, p. 35.

[SHB19] Schubert, P. D.; Hermann, B.; Bodden, E.: PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Vojnar, T.; Zhang, L. Cham: Springer International Publishing, 2019, pp. 393–410

[SHB21] Schubert, P. D.; Hermann, B.; Bodden, E.: Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis. In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Ed. by Møller, A.; Sridharan, M. Vol. 194. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 2:1–2:31.

[TW18] Töws, M.; Wehrheim, H.: Information Flow Certificates. In: *ICTAC*. Ed. by Fischer, B.; Uustalu, T. Vol. 11187. Lecture Notes in Computer Science. Springer, 2018, pp. 435–454.

[WDP14] Wiersema, T.; Drzevitzky, S.; Platzner, M.: Memory Security in Reconfigurable Computers: Combining Formal Verification with Monitoring. In: *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. 2014, pp. 167–174

[WSW13] Wonisch, D.; Schremmer, A.; Wehrheim, H.: Programs from Proofs - A PCC Alternative. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Sharygina, N.; Veith, H. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 912–927.