Report from Dagstuhl Seminar 12051

Analysis of Executables: Benefits and Challenges

Edited by

Andy M. King[1], Alan Mycroft[2], Thomas W. Reps[3], and Axel Simon[4]

1 **University of Kent, GB,** A.M.King@kent.ac.uk
2 **University of Cambridge, GB,** am@cl.cam.ac.uk
3 **University of Wisconsin – Madison, US,** reps@cs.wisc.edu
4 **TU München, DE,** Axel.Simon@in.tum.de

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 12051 "Analysis of Executables: Benefits and Challenges". The seminar had two focus groups: security engineers who need to find bugs in existing software systems and people in academia who try to build automated tools to prove correctness. The meeting of these diverse groups was beneficial and productive for all involved.

Seminar 29. January – 03. February, 2012 – www.dagstuhl.de/12051
1998 ACM Subject Classification B.2.2 Worst-case analysis, D.2.4 Formal methods, D.3.2 Macro and assembly languages, D.3.4 Debuggers and Interpreters, D.4.5 Fault-tolerance and Verification, D.4.6 Information flow controls and Invasive software, D.4.8 Modelling and prediction, D.4.9 Linkers and Loaders, F.3.2 Operational semantics and Program analysis, I.2.2 Program modification
Keywords and phrases Executable analysis, reverse engineering, malware detection, control flow reconstruction, emulators, binary instrumentation.
Digital Object Identifier 10.4230/DagRep.2.1.100
Edited in cooperation with Edward Barrett

1 Executive Summary

Axel Simon
Andy King

License Creative Commons BY-NC-ND 3.0 Unported license
© Axel Simon and Andy King

The analysis of executables is concerned with extracting information from a binary program typically, though not exclusively, with program analysis techniques based on abstract interpretation. This topic has risen to prominence due to the need to audit code, developed by third parties for which the source is unavailable. Moreover, compilers are themselves a source of bugs, hence the need to scrutinise and systematically examine executables.

Seminar topics

The theme of the analysis of executables is an umbrella term adopted for this seminar, covers, among other things, the following topics:

Except where otherwise noted, content of this report is licensed under a Creative Commons BY-NC-ND 3.0 Unported license

- specifying the semantics of native instructions, intermediate languages and the synthesis of transfer functions from blocks of instructions;
- abstract domains for binary analysis and how to combine them; type synthesis;
- control-flow graph (CFG) reconstruction, which is a prerequisite for many program analysis, and CFG matching, which is useful for detecting piracy;
- self-modifying code, characterising its semantics and detecting malware.

Chronological overview of the discussion

For practical reasons, all talks on Monday were held by the four organizers, including an overview of various known tools created by Thomas Reps and his group. His talk was followed by synthesis of transfer functions (the semantics of basic blocks) using SAT solving by Andy King, type reconstruction by Alan Mycroft and the combination of several abstract domains by Axel Simon. These rather varied topics gave a good introduction. Thomas Reps suggested that we identify common goals through a group discussion, which we could not complete on Monday due to the lack of time. Instead, we scheduled mostly industrial talks on Tuesday in order to find out about the problems that security engineers face in their everyday work and which tools they developed themselves. With this information, a group discussion on Tuesday afternoon quickly raised specific issues and their priorities: analyses must be scalable, preferably to some 12.5 billion instructions that large and vulnerable applications such as Adobe Reader are comprised of. This focus begs the question of whether we can afford a sound analysis or, as was suggested on the last day of the talk on CFG reconstruction, if an engineer can afford to work on a CFG in which not all indirect jumps are resolved precisely. In general, we should be aware of what assumptions we are making, for instance, about the correctness of CPU hardware, and possibly focus more on tools that are sound only under certain assumptions. This would still be an improvement since most security engineers nowadays even use unsound tools if they are helpful. A laudable long-term goal is the verification of a browser.

A more technical topic was the way we think about the control flow of a program, in the sense that associating a program counter address with a control flow graph node is inadequate in the presence of self-modifying code. Similarly, it is not clear what constitutes a function (due to for example, tail sharing) and how to reliably identify a function in the presence of obfuscated or optimized code that does not adhere to any standard ABI. It was pointed out that functions can have hundreds of entries with a large common body, implying that duplicating this body for each entry might create a considerable code size increase for an analysis.

To contrast the applied side of binary analysis with a theoretical view on static analysis, we scheduled the more theoretic talks on Wednesday morning. The speakers addressed how mutating malware could be classified (Roberto Giacobazzi) and how to treat memory allocated from a static array as independent heap cells (Xavier Rival). These topics gave an outlook on the challenges that lie beyond the already complicated reconstruction of the control flow graph.

Thursday and Friday featured talks mostly from the academic community who presented their current state-of-the-art. One particular debate arose on how the semantics of assembler instructions are best expressed. During an informal meeting on Thursday evening we agreed that the community would benefit from a common infrastructure to decode executable code. The way in which we should specify the semantics of native instructions was more difficult

to agree upon. Thus, we set up a mailing list to discuss a common decoder infrastructure that should be able to accommodate several platforms (say ARM and x86). The design of a decoder should feature a domain specific language that allows for a human readable specification of decoding instructions. This DSL should ideally be usable to also express the semantics of instructions, even if the various groups might want to implement their own semantic interpretation depending on their analysis needs.

Participation

In all, 42 researchers, both senior and more junior, from 10 countries attended the meeting. This high number shows the strong interest in this emerging field. The feedback from the participants was also very positive.

Directions for the future

Thus, one of the tangible outcomes is that the community set out to create a common piece of infrastructure. Beyond this, it was agreed that another seminar about the analysis of executables in two years time would be most welcome. We discussed what topics this new seminar should focus on and we distilled that malware, obfuscation, interpreters and self-modifying code should be major topics, as these constitute challenges that the community needs to address.

2 Table of Contents

Executive Summary
Axel Simon and Andy King . . . 100

Overview of Talks

A Tale of Two Tools: BEST & GIRA
Gogul Balakrishnan . . . 105

Refinement-based CFG Reconstruction from Unstructured Programs
Sebastien Bardin . . . 105

Model Checking PLC Programs
Sebastian Biallas . . . 105

On Backward Analysis in Binary Code using SAT/SMT
Jörg Brauer . . . 106

Evaluating Binary Code Diversification
Bjorn De Sutter . . . 106

Comparison, Navigation, Classification
Thomas Dullien . . . 107

Insight Framework: Yet Another Executable Binary Analysis Framework...
Emmanuel Fleury . . . 107

Fast Linear Two Variable Equalities
Andrea Flexeder . . . 107

Metamorphic Code Analysis by Abstract Interpretation
Roberto Giacobazzi . . . 108

Emulator Design, Traps and Pitfalls
Paul Irofti . . . 108

Jakstab & Alternating Control Flow Reconstruction
Johannes Kinder . . . 109

Transfer Function Synthesis at the Bit-level
Andy M. King . . . 109

Context Sensitive Analysis Without Calling Context
Arun Lakhotia . . . 109

In Situ Reuse of Functional Components of Binaries
Arun Lakhotia . . . 110

TSL: A System for Automatically Creating Analysers and its Applications
Junghee Lim . . . 110

Scalable Vulnerability Detection in Machine Code
Alexey Loginov . . . 111

Analysis of Binaries: An Industrial Perspective
Florian Martin . . . 111

PEASOUP: Preventing Exploits Against Software of Uncertain Provenance
David Melski . . . 111

Binary Code Analysis and Modification with Dyninst
Barton P. Miller 112

Decompilation, Type Inference and Finding Code
Alan Mycroft 112

A Formal ARM Model and Its Use
Magnus Myreen 113

There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code
Thomas W. Reps 113

Race Condition Detection in Compiled Programs
Andrew Ruef 114

Combining Several Analyses Into One or What Is a Good Intermediate Language for the Analysis of Executables?
Axel Simon 114

Constraint-Based Static Analysis of Java Bytecode
Fausto Spoto 114

A Method for Symbolic Computation of Abstract Operations
Aditya Thakur 115

Adversarial Program Analysis and Malware Genomics
Andrew Walenstein 115

Participants 116

3 Overview of Talks

3.1 A Tale of Two Tools: BEST & GIRA

Gogul Balakrishnan (NEC Laboratories America, Inc. – Princeton, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Gogul Balakrishnan

I will describe the BEST & GIRA tools developed at NEC Labs America.

BEST (Binary-instrumentation-based Error-directed Symbolic Testing) is a tool for finding problems in multi-threaded C/C++/Java programs. BEST uses binary-instrumentation to extract traces of execution runs, and uses SMT-based symbolic techniques to explore alternate schedules not visited during the given execution run. BEST can be used during testing to predict program failures, or during debugging to replay program failures.

GIRA (Generation of Intermediate Representation for Analysis) is a framework for analysing C++ programs. When describing GIRA, I will demonstrate that an executable compiled from C++ is very static-analysis unfriendly, and show how GIRA can alleviate the problem.

3.2 Refinement-based CFG Reconstruction from Unstructured Programs

Sebastien Bardin (CEA – Gif-sur-Yvette, FR)

License Creative Commons BY-NC-ND 3.0 Unported license
© Sebastien Bardin

We address the issue of recovering a both safe and precise approximation of the Control Flow Graph (CFG) of a program given as an executable file. CFG reconstruction is a cornerstone of safe binary-level analysis: if the recovery is unsafe, subsequent analyses will be unsafe too; if it is too rough, they will be blurred by too many unfeasible branches and instructions. The problem is tackled with a refinement-based static analysis working over finite sets of constant values. The refinement mechanism allows to adjust the domain precision only where it is needed, resulting in precise CFG recovery at moderate cost.

First experiments, including an industrial case study from aeronautics, give promising results in terms of precision and efficiency.

3.3 Model Checking PLC Programs

Sebastian Biallas (RWTH Aachen, DE)

License Creative Commons BY-NC-ND 3.0 Unported license
© Sebastian Biallas

Programmable Logic Controllers (PLCs) are control devices used in the automation industry for operating robots, machines and plants. This talk presents the ArcadePLC (Aachen Rigorous Code Analysis and Debugging Environment for PLC) framework to verify PLC programs, written in various languages used in industry.

ArcadePLC provides a model checker and static analysis to prove safety properties and aid in program understanding. PLCs usually operate in the cycling scanning mode, which consists of three atomically and repeatedly executed phases: (1) reading input variables from sensors, (2) executing the program and (3) write-back of output variables which are connected to actuators. To verify such programs, the user can specify relations of inputs/outputs for the model checker in ACTL and ptLTL logic, which are evaluated at the end of each cycle (which corresponds to the observable behaviour).

To allow for model checking larger programs, we use abstract and symbolic simulation of the program. Our key idea is to exploit the cyclic operation mode of PLCs: In the first phase, will build successors by performing symbolic execution.

For ambiguous control flow, we use this symbolic information to infer weakest preconditions on the inputs. This allows for successively refining input values until the control flow is deterministic. Then, we discard the symbolic information and store only interval and bit-set information in state space. In the second phase, we use a CEGAR technique: Possible counterexamples are analysed and – if necessary – used to further refine the state space.

We used ArcadePLC to successfully verify different libraries of function blocks used in industry.

3.4 On Backward Analysis in Binary Code using SAT/SMT

Jörg Brauer (RWTH Aachen, DE)

License Creative Commons BY-NC-ND 3.0 Unported license
© Jörg Brauer

Over the past decade, a variety of techniques have been invented that automatically compute optimal abstractions in the abstract interpretation framework. Impressive progress on decision procedures such as SAT and SMT solvers has made these techniques a practical proposition. However, it is important to note that automatic abstraction has thus far concentrated on forward abstraction.

Our presentation focuses on problems and techniques that operate in both, forward and backward direction. We identify domain-theoretic properties which explain the problems involved in backward analyses, and propose a framework based on the computational domain of Boolean formulae to circumvent these problems. Further, we report on a method that computes value-set approximations alternately in forward and backward directions. This technique allows us to reconstruct an accurate control flow graph from binary code using incremental SAT solving.

3.5 Evaluating Binary Code Diversification

Bjorn De Sutter (Ghent University, BE)

License Creative Commons BY-NC-ND 3.0 Unported license
© Bjorn De Sutter

Software diversity has been proposed as a mechanism to support renewability in a range of software protection techniques, as well as a direct defence against collusion attacks or against the automation of attack scripts. This paper evaluates the potential of software diversity to protect against collusion attacks on security patches, such as the attacks commonly referred

to as "exploit Wednesday" attacks. Those attacks build on patches released on "Microsoft patch Tuesday" and rely on the fact that security fixes are easy to identify in undiversified software. This paper evaluates the feasibility of adapting the (semi-)automated attacks described in literature to diversified software, for a range of diversifying transformations of different strengths. We found that all existing tools can easily be thwarted, thus making the automation of the existing attacks on diversified software infeasible.

3.6 Comparison, Navigation, Classification

Thomas Dullien (Google Switzerland – Zürich, CH)

License Creative Commons BY-NC-ND 3.0 Unported license
© Thomas Dullien

This talk discusses the algorithms and ideas used in BinDiff, BinNavi, VxClass which were tools distributed by zynamics prior to the acquisition by Google.

3.7 Insight Framework: Yet Another Executable Binary Analysis Framework...

Emmanuel Fleury (Université Bordeaux, FR)

License Creative Commons BY-NC-ND 3.0 Unported license
© Emmanuel Fleury

The Insight framework is a executable binary analysis framework for UNIX platforms and aiming at validation, verification and reverse-engineering binaries. The framework comes with a proposal of a machine-code independent intermediate representation that allows manipulation (e.g. for deobfuscation).

3.8 Fast Linear Two Variable Equalities

Andrea Flexeder (TWT GmbH, DE)

License Creative Commons BY-NC-ND 3.0 Unported license
© Andrea Flexeder

We present a novel interprocedural analysis of linear two-variable equalities which has a worst-case complexity of $\mathcal{O}(nk^4)$, where k is the number of variables and n is the program size. The analysis can be applied for identifying local variables and thus for interprocedurally observing stack pointer modifications as well as for an analysis of array index expressions, when analysing low-level code.

3.9 Metamorphic Code Analysis by Abstract Interpretation

Roberto Giacobazzi (Università degli Studi di Verona, IT)

License Creative Commons BY-NC-ND 3.0 Unported license
© Roberto Giacobazzi

Metamorphic code includes self-modifying semantics-preserving transformations to exploit code diversification. The impact of metamorphism is growing in security and code protection technologies, both for preventing malicious host attacks, e.g., in software diversification for IP and integrity protection, and in malicious software attacks, e.g., in metamorphic malware self-modifying their own code in order to foil detection systems based on signature matching. In this paper we consider the problem of automatically extracting metamorphic signatures from metamorphic code. We introduce a semantics for self-modifying code, later called phase semantics, and prove its correctness by showing that it is an abstract interpretation of the standard trace semantics. Phase semantics precisely model the metamorphic code behaviour by providing a set of traces of programs which correspond to the possible evolutions of the metamorphic code during execution. We show that metamorphic signatures can be automatically extracted by abstract interpretation of the phase semantics. In particular, we introduce the notion of regular metamorphism, where the invariants of the phase semantics can be modelled as finite state automata representing the code structure of all possible metamorphic changes of a metamorphic code, and we provide a static signature extraction algorithm for metamorphic code where metamorphic signatures are approximated in regular metamorphism.

3.10 Emulator Design, Traps and Pitfalls

Paul Irofti (FileMedic Ltd., PL)

License Creative Commons BY-NC-ND 3.0 Unported license
© Paul Irofti

During the last two years I've been researching the field of dynamic analysis in regards to emulating obfuscated and/or malevolent binaries. The result is an emulator that translates code blocks of binary samples from different platforms (operating systems and machine types) into an intermediate representation where information retrieval, data analysis and behaviour observations are made. After a code block is compiled and executed on the host platform and the entire environment is updated accordingly. Unless a verdict has been reached, a new cycle begins.

During the Dagstuhl Seminar I will present in-depth the design of this emulator and exchange ideas with people involved in similar activities.

3.11 Jakstab & Alternating Control Flow Reconstruction

Johannes Kinder (EPFL – Lausanne, CH)

License Creative Commons BY-NC-ND 3.0 Unported license
© Johannes Kinder

Unresolved indirect branch instructions are a major obstacle for statically reconstructing a control flow graph (CFG) from machine code. If static analysis cannot compute a precise set of possible targets for a branch, the necessary conservative over-approximation introduces a large amount of spurious edges, leading to even more imprecision and a degenerate CFG.

We propose to leverage under-approximation to handle this problem. We provide an abstract interpretation framework for control flow reconstruction that alternates between over- and under-approximation. Effectively, the framework imposes additional preconditions on the program on demand, allowing to avoid conservative over-approximation of indirect branches. We implemented the framework on top of our binary analysis tool Jakstab and present very promising results from using only constant propagation and a single concrete execution trace per target.

3.12 Transfer Function Synthesis at the Bit-level

Andy M. King (University of Kent, GB)

License Creative Commons BY-NC-ND 3.0 Unported license
© Andy M. King

In this talk we review how concrete semantics of blocks, represented as SAT or SMT instances, can be used to distil transfer functions that operate over systems of congruences and octagons. The reoccurring idea is to repeatedly solve an instance, collect different solutions, and then merge them to derive a summary for a block as a whole. We show how this technique can be applied to deobfuscate blocks to recover their meaning as well as derive transfer functions that can be composed so as to derive invariants from binary code.

3.13 Context Sensitive Analysis Without Calling Context

Arun Lakhotia (University of Louisiana – Lafayette, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Arun Lakhotia

Classic methods of interprocedural analysis are based on reachable paths defined over interprocedural control flow graph (ICFG). Adpating these methods to binaries require static identification of procedure 'call' and 'ret' instructions. There are many instances when a binary may not use such instructions to call (or return from) a procedure, such as, with tail-merge or body-merge operations performed by optimizing compilers or obfuscations used by malware.

We present a method to perform context-sensitive analysis using a 'stack graph' instead of 'call graph'. This method removes the need for identifying atomic instructions that modify the stack as well as transfer control. Instead our method requires only the ability to statically identify statements that modify the stack pointer.

3.14 In Situ Reuse of Functional Components of Binaries

Arun Lakhotia (University of Louisiana – Lafayette, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Arun Lakhotia

A complex binary is a composition of many behaviours. Access to these behaviours is provided through a user interface chosen by the programmers. There are times when one may need to access some part of the binary's behaviour or access its behaviour in ways that were not imagined by the original designers. One way to achieve this is to replicate the specific behaviour of the binary in another, independent program and use it. Such ex situ methods can be challenging, since they require creating code that can be independently compiled.

We present a method to use the functionality of the binary in situ, that is, directly within the binary without physical extraction. The architecture consists of three parts: a LEFC (logical extraction of functional component) identifier, a LEFC compiler, and a LEFC execution monitor. A functional component is defined as an entry point, a collection of exit points, a list of parameters (registers, locations), pre-condition state of the program required for the FC to behave well, and types of the parameters. The extraction of this information may be done manually or automatically. The LEFC compiler compiles this descriptor into a library, that provides a standard function call interface to the FC. To reuse the FC, a programmer links with this library. When the function is invoked, the LEFC Monitors executes the original program and communicates with its process to executes the required code directly in the program's address space.

We discuss a prototype implementation of this concept using OllyDbg. The LEFC compiler creates script for OllyDbg's scripting plug-in. A user may use these scripts to access an FC.

3.15 TSL: A System for Automatically Creating Analysers and its Applications

Junghee Lim (University of Wisconsin – Madison, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Junghee Lim

In this talk, I presented the design and implementation of system, called TSL that provides a systematic solution to the problem of creating retargetable tools for analysing machine code. TSL is a meta-tool; a tool generator that automatically creates different abstract interpreters for machine code instruction sets. TSL advances the state of the art in program analysis by providing a YACC-like mechanism for creating the key components of machine code analysers from a description of the concrete operational semantics of a given instruction set. TSL automatically creates implementations of different abstract interpreters for the instruction set.

I also briefly talked about various application tools developed via the TSL system.

3.16 Scalable Vulnerability Detection in Machine Code

Alexey Loginov (GrammaTech Inc. – Ithaca, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Alexey Loginov

This talk describes the design and implementation of a scalable and precise tool for detecting vulnerabilities in machine code. The talk presents project goals, an overview of the tool architecture, the evaluation strategy for the tool, as well as how the evaluation strategy evolved as we gained experience during broader application of the tool. The talk will conclude with a discussion of a few challenges that may require the combined efforts of this community.

3.17 Analysis of Binaries: An Industrial Perspective

Florian Martin (AbsInt – Saarbrücken, DE)

License Creative Commons BY-NC-ND 3.0 Unported license
© Florian Martin

In safety-critical systems a worst case execution time (WCET) analysis is vital, as it is the prerequisite for schedulability analysis. aiT is a sound WCET analyser, which is available for many different target processors. As the execution time is influenced greatly by the compiler and even can be influenced by the linker, the analyser works on fully linked binaries.

This talk will present the basic architecture of aiT. It will discuss some of the challenges and benefits which arise from analyzing executables, and the methods to cope with them.

3.18 PEASOUP: Preventing Exploits Against Software of Uncertain Provenance

David Melski (GrammaTech Inc. – Ithaca, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© David Melski

We present ongoing research on PEASOUP, a technology that enables the safe execution of software executables of uncertain provenance. PEASOUP (Preventing Exploits Against Software Of Uncertain Provenance) provides multi-level protection against the exploitation of multiple vulnerability classes. PEASOUP's operation is divided into an offline analysis phase and an online monitoring phase. The analysis phase builds an IR for the subject executable, produces multiple hardened, diversified variants of the subject executable, and tests the variants for resistance to attack and conformance with the original executable. The execution monitoring stage selects a variant of the subject, transforms the subject into the variant on demand during execution, and monitors the runtime execution for attempted exploits. This work is sponsored by the US Air Force Research Labs.

3.19 Binary Code Analysis and Modification with Dyninst

Barton P. Miller (University of Wisconsin – Madison, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Barton P. Miller

The Dyninst suite of toolkits provides a platform on which to build a wide variety of tools for operating on binary programs. Such tools include those for debugging, tracing, performance profiling, code optimization, testing, modelling, and cyber forensics.

Dyninst provides both control and data flow analyses of code, including live register analysis and slicing. The control flow analysis will identify functions, loops, basic blocks and instructions. As part of this analysis, Dyninst identifies (and can use for instrumentation) function entry and exit points; call sites; and loop entry, exit and body. Analysis occurs both at start time and during execution as new code is discovered (loaded dynamically, unpacked, or found based on tracking obfuscated control flow operations).

Instrumentation and modification of the code is based on patching the new operations into the code. Only the code that is being instrumented or modified is effected. Dyninst is a major customer of its own analyses, using them to generate efficient instrumentation code. Code modification as done in terms of editing the program's control flow graph and updating individual instructions in basic blocks. All instruction-level code changes are in terms of an abstract syntax tree representation, so are platform independent and portable.

For analysing and instrumenting malicious code, Dyninst has the ability to detect and deactivate defensive checks, and capture obfuscated control flow such as those based on return address manipulation, exceptions, run-time unpacking of code, and instruction overwriting. This defensive mode of Dyninst has been tested with code generated by most of the popular code packers and obfuscators.

Dyninst is actually a suite of toolkit libraries that can be used separately or in combination. These libraries support such functionality as code parsing, instruction cracking, symbol table reading and modifying (a surprisingly complex and tricky package), dataflow analysis and symbolic execution, code patching, dynamic code generation, process control, stack walking, and a C-like language interface to instrumentation code specification.

Dyninst will operate on executables and libraries, both statically and dynamically linked). While Dyninst operates happily on stripped binaries, it will also make best use available symbols (both static and dynamic) and debugging information. Supported platforms for Dyninst include x86 (32 and 64 bit) on Linux and Windows, Power (32 and 64 bit) on Linux and BlueGene.

Dyninst is also a platform for research into new techniques in program forensics (determining the provenance and authorship of the binary), vulnerability assessment of the code, and fault diagnosis.

3.20 Decompilation, Type Inference and Finding Code

Alan Mycroft (University of Cambridge, GB)

License Creative Commons BY-NC-ND 3.0 Unported license
© Alan Mycroft

Decompilation is a mechanism for attempting to understand lower-level code by reconstructing source code of similar functionality. For type-unsafe languages such as C this is inherently

problematic since C's 'undefined behaviour' allows return addresses etc. to be modified in a way which cannot be portably expressed as C source by a decompiler.

We highlight this decompiler choice between functionality and beauty and note that it occurs at all levels in the decompiler pipeline from executable to binary payload to assembler source to high-level code and is particularly an issue in malware.

The second topic notes that many techniques for compilation and decompilation are common, e.g. SSA removes aliasing performed by register allocation. In particular, for assembler code in SSA, we show how a variant of Hindley-Milner type reconstruction can construct C-level types, including recursive structs, ab initio.

3.21 A Formal ARM Model and Its Use

Magnus Myreen (University of Cambridge, GB)

License Creative Commons BY-NC-ND 3.0 Unported license
© Magnus Myreen
Joint work of Fox, Anthony; Sewell, Thomas; Klein, Gerwin

I presented a formal model of the ARM ISA developed by Anthony Fox. This model has its roots in a project on hardware verification, has been extensively tested and covers all current versions of the ARM ISA: ARMv4-v7.

I also showed how I've used this model in proofs inside the HOL4 theorem prover.

My main tool is a proof-producing decompiler which takes machine code (e.g. ARM) and provides the user with a concise functional description of the machine code.

This tool has been used in an extension of the L4.verified project which proved functional correctness of the seL4 microkernel.

3.22 There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code

Thomas W. Reps (University of Wisconsin – Madison, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Thomas W. Reps

Computers do not execute source code programs; they execute machine code programs that are generated from source code. Consequently, some of the elements relevant to understanding a program's capabilities and potential flaws may not be visible in its source code. The differences in outlook between source code and machine code can be due to layout choices made by the compiler or optimizer, or because transformations have been applied subsequent to compilation (e.g., to make the code run faster or to insert software protections).

The talk discussed the obstacles that stand in the way of using static, dynamic, and symbolic analysis to understand and verify properties of machine-code programs. Compared with analysis of source code, the challenge is to drop all assumptions about having certain kinds of information available (variables, control-flow graph, call-graph, etc.) and also to address new kinds of behaviours (arithmetic on addresses, jumps to "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream, self-modifying code, etc.). In addition to describing the challenges, the talk will also describe what can be done about them.

3.23 Race Condition Detection in Compiled Programs

Andrew Ruef (University of Maryland – College Park, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Andrew Ruef

Race conditions in multi-threaded programs are especially troublesome. They can manifest as deadlocks, faults, or semantic errors in program function. The nondeterminism inherent in multi-threaded programs presents challenges to testing and verifying them, especially once compiled. We present some approaches to use program rewriting to attempt to identify race conditions in compiled applications, without the assistance of any symbol information or user assistance. These systems are intended to increase the ability of quality assurance and allows developers to locate and reproduce concurrency errors in multi-threaded programs.

3.24 Combining Several Analyses Into One or What Is a Good Intermediate Language for the Analysis of Executables?

Axel Simon (TU München, DE)

License Creative Commons BY-NC-ND 3.0 Unported license
© Axel Simon

The implementation of a static analysis is a complex undertaking when several domains should be combined to yield a more precise result. We contrast the Astree approach of implementing mostly a partially reduced cardinal product versus using only functor domains (which we interpret as reduced cardinal power domains). We illustrate how affine equations, congruences and intervals can be combined this way, thereby requiring less communication and, more importantly, a simpler communication infrastructure. The advantage of functor domains is that the API of a domain can change. However, for software engineering reasons it is sensible to settle for a few APIs between domains since then an analysis is flexible in re-arranging domains. We identify four APIs (and thus intermediate languages) that we use to address the analysis of executables including the treatment of wrapping of finite integer arithmetic.

3.25 Constraint-Based Static Analysis of Java Bytecode

Fausto Spoto (Università degli Studi di Verona, IT)

License Creative Commons BY-NC-ND 3.0 Unported license
© Fausto Spoto

I will present the constraint-based static analysis technique implemented in the Julia analyser for Java and Android. Examples will be taken from field initialization analysis and reachability analysis between program variables. I will conclude with future developments and open problems.

3.26 A Method for Symbolic Computation of Abstract Operations

Aditya Thakur (University of Wisconsin – Madison, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Aditya Thakur

In 1979, Cousot and Cousot gave a specification of the best (most-precise) abstract transformer possible for a given concrete transformer and a given abstract domain. Unfortunately, their specification does not lead to an algorithm for obtaining the best transformer. In fact, algorithms are known for only a few abstract domains.

Motivated by this problem, we developed a parametric framework that, for a given abstract domain A and logic L, computes increasingly better abstract values in A that over-approximate the set of states defined by an arbitrary formula in logic L. Because the method approaches the most-precise abstract value from "above", if the computation takes too much time it can be stopped to yield a sound answer. For certain domains and logics, the framework is capable of computing the most-precise abstract value in the limit.

Our framework can be used to compute the best abstract transformer for a given abstract domain and concrete transformer represented by a formula in L. We describe instantiations of our framework for well-known abstract domains, such as intervals, polyhedra, and affine relations over bit-vectors.

3.27 Adversarial Program Analysis and Malware Genomics

Andrew Walenstein (University of Louisiana at Lafayette, US)

License Creative Commons BY-NC-ND 3.0 Unported license
© Andrew Walenstein

Three challenges for binary analysis are presented. One challenge is that of robustness of analysis, and an experiment is reported that illustrates how fusing multiple tracer outputs can yield improved automated classification.

Another challenge is of fair evaluation of robustness, and an experiment is reported that illustrates how authentic (wild) malware are likely to be poor tests of the robustness of an analysis since the analysis is not being targeted.

The final challenge presented is that of malware relationship recovery. A model-driven evaluation of the different ways in which malicious files can be derived suggests complications for relationship recovery that may be surprising to some.

Participants

- Gogul Balakrishnan
NEC Lab. America, Inc. – Princeton, US
- Sébastien Bardin
CEA – Gif-sur-Yvette, FR
- Edward Barrett
University of Kent, GB
- Sebastian Biallas
RWTH Aachen, DE
- Jörg Brauer
RWTH Aachen, DE
- Doina Bucur
INCAS3, NL
- Mihai Christodorescu
IBM TJ Watson Res. Center – Hawthorne, US
- Bjorn De Sutter
Ghent University, BE
- Thomas Dullien
Google Switzerland – Zürich, CH
- Emmanuel Fleury
Université Bordeaux, FR
- Andrea Flexeder
TWT GmbH, DE
- Roberto Giacobazzi
Univ. degli Studi di Verona, IT
- Sean Heelan
Immunity Inc., US
- Paul Irofti
FileMedic Ltd., PL
- Johannes Kinder
EPFL – Lausanne, CH
- Andy M. King
University of Kent, GB
- Arun Lakhotia
Univ. of Louisiana – Gifette, US
- Jerome Leroux
Université Bordeaux, FR
- Junghee Lim
University of Wisconsin – Madison, US
- Alexey Loginov
GrammaTech Inc. – Ithaca, US
- Florian Martin
AbsInt – Saarbrücken, DE
- David Melski
GrammaTech Inc. – Ithaca, US
- Bogdan Mihaila
TU München, DE
- Barton P. Miller
University of Wisconsin – Madison, US
- Martin Murfitt
Trustwave Ltd., London, GB
- Alan Mycroft
University of Cambridge, GB
- Magnus Myreen
University of Cambridge, GB
- Michael Petter
TU München, DE
- Thomas W. Reps
University of Wisconsin – Madison, US
- Xavier Rival
ENS – Paris, FR
- Edward Robbins
University of Kent, GB
- Daniel Roelker
DARPA – Arlington, US
- Andrew Ruef
University of Maryland – College Park, US
- Alexander Sepp
TU München, DE
- Holger Siegel
TU München, DE
- Axel Simon
TU München, DE
- Fausto Spoto
Univ. degli Studi di Verona, IT
- Aditya Thakur
University of Wisconsin – Madison, US
- Christopher Vick
Qualcomm Corp.R&D – Santa Clara, US
- Aymeric Vincent
Université Bordeaux, FR
- Andrew Walenstein
University of Louisiana – Lafayette, US
- Florian Zuleger
TU Wien, AT