Refinement Types in Practice: A Survey of Modern Applications and Research Trends

Matthew Richard
Queen's University
Kingston, Ontario, Canada
23brlv@queensu.ca

Nicholas Dionne
Queen's University
Kingston, Ontario, Canada
24kg1@queensu.ca

Yannick Abouem
Queen's University
Kingston, Ontario, Canada
24hd7@queensu.ca

Abstract

Refinement types extend traditional type systems by incorporating logical predicates, which allow developers to express or enforce more accurate and precise program properties at compile time. This expressiveness helps with detecting common software errors early on in development; these errors can include things such as invalid array indexing, division by zero, and security vulnerabilities. This paper is a survey in which we explore the utilization of refinement types across many differing domains, and in this case, those domains are cybersecurity, robotics, and data visualization. We will present each of these domains through a recent academic paper that demonstrates how refinement types can be utilized to improve things such as software correctness, safety, and robustness, as well as many others in each domain. Through this analysis, we'll be able to highlight the versatility and importance of refinement types in software development as well as provide an introduction and overview of the role refinement types play in computer science.

1 Introduction

For over one hundred years, type theory has been a foundational concept not only for mathematics, but also computer science [5]. As computer science evolved from theoretical computing and logic to development and programming, type theory started being implemented as a method of providing structure to programs [5]. However, traditional type systems often lack the ability to handle more complex program properties and constraints [5]. Refinement types were introduced to address this gap, providing a solution to handle these complex properties more effectively [3].

Refinement types are an extension of traditional typing systems through the application of constraints derived from logical predicates [3]. These constraints allow for a program's behaviour to be more precisely and correctly modeled [3]. Their ability to have software systems properly defined and have behaviour enforced is what makes refinement types so important [3]. The use of refinement types can eliminate common software errors such as buffer overflows or runtime errors like division by zero [3]. As a result, refinement types have become a widely adopted technique for software design and development.

In this paper, we provide a comprehensive survey of refinement types, examining modern research and applications through the lens of three academic papers in different domains. Through this survey, we aim to provide an understanding of what refinement types are and how they work, showcase the advantages of refinement types to understand their practical use, highlight recent academic research applying refinement types, and form insights on refinement type research based on the academic papers. Through both the overview and the discussion of modern research, this survey provides a complete context for refinement types.

Countless literature discusses the theoretical concepts of refinement types. Additionally, academic papers are produced yearly proposing novel ways to implement refinement types. However, few surveys compile research to showcase the breadth of refinement types' real-world applications in a modern context. Our survey addresses this by analyzing three novel applications across different domains: cybersecurity, robotics, and data visualization.

This survey directly connects to the key concepts in CISC 465/865, Semantics of Programming Languages. In this paper, we survey refinement types. As previously discussed, refinement types are a direct development of type theory and typing systems. Type theory forms the foundation of our course. Starting from the first week where type theory was introduced, every following week built upon the last, delving deeper and deeper into type theory and the semantics of programming languages. While not directly discussed in this course, refinement types only continue to build upon every lecture and our studies into type theory and its related concepts. Overall, throughout our survey, we showcase how the concepts taught in Semantics of Programming Languages can be applied in real world scenarios.

The following sections provide context and detail for our work. We begin by providing important background information on refinement types. We then discuss three modern academic papers focusing on the application of refinement types. This is followed by our concluding remarks, where we summarize the key insights and discuss the future of refinement types.

2 Background

The term "refinement types", first coined by Freeman and Pfenning in 1991, describes types that posses a logical predicate which provides restrictions on the values a variable of that type can assume [3]. An advantage of refinement type systems is that they ensure values respect the predicates specified by their type at compile-time [3]. This enables the compiler to detect programming and logical errors and prevent common pitfalls such as accessing array at invalid indices, division by zero , and more. For example, to prevent a division by zero, a programmer can define a type `nonzero` that given an integer value v it must render the predicate $(v > 0) \vee (v < 0)$ true. Thus, at compile time any variable of type `nonzero` that contain a value $v = 0$ will produce an error.

Refinement types can not only be applied to simple types such as integers, but we can also apply them to functions, arrays, and more complex user defined data structures to specify the correct functioning of the program [3]. A refinement type system essentially bridges the gap between formal verification and the implementation [3]. Commonly, refinement types systems use SMT solvers to verify the predicates.

Some examples of languages that implement refinement types include Liquid Haskell, a tool that extends Haskell's type system with refinement types [6], Rust, ML [2], TypeScript [7] and Scala.

3 Papers

In this section, we summarize three research papers. Each paper explores the usage of refinement types in a different domain of computer science. The first paper focuses on cybersecurity, the second focuses on robotics and the third focuses on data visualization.

3.1 Paper 1: "Cache Refinement Type for Side-Channel Detection of Cryptographic Software"

Presented in 2022, "*Cache Refinement Type for Side-Channel Detection of Cryptographic Software*" by authors Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang [4] demonstrates an application of refinement types in cybersecurity. This paper was published at the ACM Conference on Computer and Communications Security (CCS) [4]. The paper presents CaType, a novel tool utilizing refinement types designed to detect cache-based side-channel vulnerabilities in cryptographic software [4].

Refinement types are effective tools for verifying correctness, safety, and security in software systems. In cybersecurity, their use has grown for detecting more subtle vulnerabilities at the lower levels of a system [4]. This can be seen in the paper with the CaType tool for detecting side channel vulnerabilities in cryptographic software like OpenSSL or Libgcrypt using RSA, ElGamal, and ECDSA algorithms [4].

3.1.1 Problem. The problem the paper's authors have identified and want to address is side-channel attacks [4]. Side-channel attacks exploit unintended information leakage due to system behaviors. In this paper, the authors address specifically cache-based side-channel attacks [4]. These attacks can occur when cache behavior deviates based on secret data (cryptographic keys, etc) [4]. Side-channel attacks essentially allow for attackers to infer secret (cryptographic keys, etc) or sensitive information (data) by observing things such as timing differences, cache hits and misses, or patterns in how memory is accessed [4].

In cryptography, secrets are extremely important and should remain hidden by all means. However, some subtle differences in cryptographic software execution, such as which path to take based on a secret or the utilization of a lookup table using a secret index, can leak this information in the process and this is the aforementioned exploitation method for side channel attacks [4]. Not all is lost though, there are some defenses for side-channel attacks at a software level. Two of the more popular defenses are Constant-Time Coding Practices and Blinding Techniques [4]. Constant-Time Coding Practices help to defend against side-channel attacks by writing code whose execution time is not dependent on secret data, to avoid attackers exploiting the execution time variations to uncover secret data [4]. Blinding Techniques help to defend against side-channel attacks by randomizing the behaviors of cryptographic computations to hide secret information [4]. Both of these methods are applied in various forms, but ultimately the authors find it hard to ensure or detect that defense strategies such as these are correctly implemented in complex cryptographic libraries [4].

The authors propose a tool called CaType, which utilizes refinement types as a potential solution to the aforementioned issue [4]. However, there are already other tools like CacheAudit and CaSym, which are static analysis and symbolic execution tools, respectively. So how do the authors justify their work and the need for the application of a refinement type tool in this area [4]? The authors claim that although effective, these tools suffer from limitations, firstly in terms of scalability, as they are computationally expensive to run on real-world cryptographic software [4]. Secondly, the authors claim that these tools produce a high number of false positives due to imprecise modeling of memory and or cache behaviors [4].

3.1.2 Solution. The proposed solution by the authors is CaType, a refinement type tool for cache-based side-channel attack vulnerability detection [4]. This tool is supposed to not only address the problem, but do it better than previous tools, with the author's acclaimed limitations [4]. CaType utilizes an alternate approach to previous tools based on refinement type inference at the binary level [4]. CaType delves into the x86 executables of cryptographic libraries (OpenSSL, Libgcrypt, etc) and does bit-level tracking of secret data through the refinement type system set up with CaType [4]. By doing so, Catype captures both the direct use of secret data and the utilization of blinding to protect from cache-based side-channel attacks [4].

Diving a little deeper into the paper's application of refinement types [4], CaType has each variable in the program assigned to a type with a predicate describing the security property [4]. An example would be a 32-bit variable k (k is typically a secret key in cryptography). This variable would likely be assigned the type $uint32v : SDD$ by CaType which means secret dependent value [4]. There are also four other types that CaType introduces and utilizes. These types are:

- URA (Uniformly Random): Data that's been randomized by blinding [4].
- SID (Secret-Independent): Non-secret data [4].
- WRA (Weakly Random): Imperfectly randomized data [4].
- CST (Constant): Hard-coded constants [4].

These allow for CaType to effectively model the flow of information, tracking how secrets propagate through the cryptographic software when both explicitly and implicitly assigned [4].

3.1.3 Results.

An important aspect of developing tools such as CaType for security is testing the tool thoroughly, especially if it's among the first times a technique such as refinement types has been applied to cache side channel attacks in a cryptographic setting [4]. The authors do in fact test CaType quite thoroughly on several well-known industry-level cryptographic libraries. These include both OpenSSL and Libgcrypt, where they analyze these libraries' implementations of RSA, ElGamal, and ECDSA using CaType [4]. Another point is that these libraries also apply blinding and Constant-Time Coding practices, which can, in some cases, adversely affect tools for detecting vulnerabilities in cryptographic software, which makes these all the more important for real-world application of CaType and its findings [4]. In the end, CaType was able to detect all known side-channel vulnerabilities in the tested libraries, identify previously unknown vulnerabilities that remained undetected by other tools. Moreover, CaType was able to significantly reduce false positives compared to symbolic execution and taint-tracking approaches. Lastly, CaType was able to analyze the libraries 16x faster than CacheD and 131x faster than CacheS, both tools for cache-based side-channel attack vulnerability detection [4].

Lastly, and maybe the most impactful aspect of CaType and the utilization of refinement types in this setting, other than the significant speed up, is URA [4]. URA allows for CaType to mark identified instances of blinding and other randomization defenses implemented in cryptographic libraries, which reduces the number of false positives significantly when identifying information leakage [4]. The authors mention that other tools were ineffective in this regard, meaning they were unable to model these defenses and take care of the resulting false positives through their methods [4].

CaType, as introduced by the authors of the paper, shows the practical utility and effectiveness of refinement types in cybersecurity. As CaType has the ability to more readily scale to real-world code and even model the complex defense strategies involved to help avoid false positives better than most other tools. Overall, refinement types are a useful and yet effective tool for analyzing and improving upon secure systems.

3.2 Paper 2: "Synchronous Programming and Refinement Types in Robotics: From Verification to Implementation"

The second paper chosen for the survey is "*Synchronous Programming and Refinement Types in Robotics: From Verification to Implementation*" by Chen et al [1]. This paper was first presented in 2022 at the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems [1].

The authors identify that there has been an increased use of cyber-physical systems (CPS) in daily life [1]. CPS are systems that combine both computational and physical components [8]. For example, smart homes, autonomous vehicles, and even robots [8]. In this paper, the authors focus on robot CPS, noting that their potential to cause physical harm makes it vital to ensure that they are properly designed [1]. Formal verification, a method of proving system correctness with mathematical proofs, is often the solution for ensuring the proper design of a system [1]. However, formal verification languages are often complex, and need to first be translated to be executed, which can cause new issues in the system [1]. Conversely, languages designed for CPS execution typically do not allow for formal verification [1]. As a result, there is a clear gap for languages that can both implement formal verification, and be executed directly on a CPS without translation. To fill in this gap, Chen et al. introduce MARVeLus [1].

MARVeLus (Method for Auto-mated Refinement-type Verification of Lustre) is a novel language created by the authors to solve the problem of there not being a language that combines formal verification and CPS execution [1]. The authors define MARVeLus as an extension of the synchronous programming language Zélus [1]. Synchronous programming languages divide periods of time into different blocks called steps; at each step, the program performs its computations and updates values simultaneously [1]. MARVeLus is further characterized as having execution capabilities like the language Lustre, combined with refinement type formal verification similar to Liquid Haskell [1]. By extending the capabilities of Zélus, MARVeLus code that is written is translated to OCaml, which allows for the MARVeLus extensions to both compile and execute directly on a CPS [1]. To run the OCaml code on the robot CPS, the Lightweight Communications and Marshalling (LCM) protocol is used for connecting the verified program to the robot's sensors and actuators in real time [1]. Ultimately, the proposed language, MARVeLus, allows for formal verification and CPS execution, specifically for robots, all in one language.

Chen et al. ground their work by applying MARVeLus to a real world robotics problem, adaptive cruise control (ACC) [1]. ACC is a more advanced version of regular cruise control in cars; it offers the traditional cruise control functions, with the addition of automated collision detection in your lane [1]. With this, automated breaking to prevent collisions is a major part of this real world problem that can be modeled through formal verification. To model a simplified version of this problem, the system has two main components, the follower vehicle in which the software is controlling, and a fixed obstacle that the vehicle must avoid, known as the lead vehicle [1]. The software has knowledge of the follower and lead vehicles locations, and the distance from the follower vehicle to the lead vehicle [1]. Additionally, the software has full control of the follower vehicle, being able to modify speed and braking [1]. A set of mathematical formulas are made to represent the model [1]. These formulas measure position of the follower vehicle, trigger braking to avoid collision, and ensure that the follower never passes the leader [1].

Refinement types are not only the focus of our survey, but they are also vital to authors work in this paper. As previously mentioned, MARVeLus contains formal verification through refinement types [1]. More specifically, this work implements liquid types [1]. Liquid types are form of refinement types that restrict predicates to formulas so that they can be verified using a Satisfiability Modulo Theory (SMT) solver [1]. To implement refinement types in a synchronous language, the others develop a set of syntax typing rules which record values across all time [1]. Their rules are unique in the fact that they allow for branching, which is the concept of `if x, then y, else z` [1]. Through the syntax and typing rules of MARVeLus which are built upon the refinement types, the software executions are constrained [1]. To verify that the refinement types conditions are validated, all software is passed through the Z3 SMT Solver [1]. If the solver can verify the conditions, the software is executed by the CPS [1].

To implement the proposed typing syntax and rules, Chen et al. design a program using MARVeLus [1]. Each constraint is modeled in their code. While the formal verification of software is one part of their work, the remaining part falls in the physical execution of a CPS [1]. Acting as the follower vehicle in the ACC domain previously outlined, the authors use a M-Bot Mini [1]. The M-Bot Mini is a programmable wheeled robot vehicle, which contains a LIDAR sensor, that is typically used for educational purposes [1]. In testing purposes, a setup is used where the program is first verified on a desktop computer, compiled to OCaml, and the OCaml files are then sent to the robot to be executed [1].

The actual testing of MARVeLus in the ACC domain was completed both in a simulation and in real life [1]. The ACC system, implemented with MARVeLus, was successful in braking before collision across all tests completed through simulation and in real life [1]. The behaviours of the robot in each test are similar across testing implementations, that is, both the position and velocity results are very close [1]. The authors do note that the real life robot stops 0.2 meters before the simulated robot, which they contribute to a slow LIDAR sensor [1]. Overall, experimental results from Chen et al. prove the validity of MARVeLus as a formal verification and execution language.

Chen et al. note several directions for future work that they would like to pursue [1]. The first direction noted is implementing simulations for automata and hybrid models [1]. Next, they discuss the importance to later implement end-to-end safety [1]. Lastly, the authors outline wanting to both improve the hardware implementation, and expand the usability of MARVeLus to other robot platforms [1].

Overall, MARVeLus accomplishes the authors goal of filling in the gap for an all in one language that implements formal verification and can be executed directly by a CPS. Through their work, Chen et al. not only show the importance of refinement types in formal verification, but also show that refinement types can have meaningful real life applications.

3.3 Paper 3: "Refinement Types for Visualization"

In this 2024 paper, Xia et al. seek to create a tool for data utilization by using a synthesis algorithm combined with refinement types. The result of their research is a domain specific language (DSL), called Calico. In this section we will explore their approach, the design of Calico, and the results they obtained.

Calico works by synthesizing a program composed of a table program and a visual program given an input table and a visual trace [9]. Despite these two components, the paper mostly focuses on the table program. A table program is a sequence of operators that are applied over the input table and produces an output table [9]. The output table is then used to produce the chart or visualization requested by the user. The table program is synthesized from the visual trace, which, unlike other tools, is encoded by Calico as a refinement type. Other tools require the user to provide concrete values in the visual trace in order for the program to be synthesized, while Calico accepts constraints as part of the trace to build refinement types that the output table will have to abide by [9]. This allows the user to avoid the tedious and error-prone process of determining concrete values for the visual trace [9].

Another technique used in the creation of the program is bidirectional analysis. This technique is used to determine if a generated partial program is feasible or should be discarded [9]. This is done by approximating the effect that applying or reversing the application of an operator will have on the type of the table and create a type consistency constraint [9].

3.3.1 Calico's Table Transformation Language. The language the authors present in this paper was designed to be similar to other table construction languages such as SQL and R [9]. As stated earlier, the table transformation language is a set of transformations applied over an input table. The operations that are used by Calico are the following:

- `select`: keep selected columns and drop the rest.
- `filter`: keep the rows that satisfy a certain predicate.
- `spread`: uses the elements of a column as keys and the elements of another as values and constructs two rows.
- `gather`: inverse of `spread`.
- `summarize`: construct groups from a column of keys and aggregate these groups using an aggregation operation.

3.3.2 Calico's Type System. Calico's refinement type system is composed by the following types:

- The table type: a collection of column and row types.
- The column type: pairs a column name to a refinement type.
- The row type: has a sequence of cell types which contain a row name and a refinement type that describes the contents of the cell.
- A refinement type: consists of a base type and a logical quantifier.

Calico also possesses a provenance predicate, used to track the set of labels necessary to compute a specific table, as well a related-operator predicate that tracks the operators used to compute the current cell or column [9].

3.3.3 The Synthesis Algorithm and Bidirectional Analysis. The algorithm used by Calico accepts arguments such as an input table, the desired type of the output table, and the maximum length of the desired program [9]. The algorithm will first determine the type of the input table and generates *partial programs* of the desired length [9]. The algorithm will then enter a loop and fill each partial program until the first viable is found [9]. In the loop, the algorithm will approximate the functioning of the algorithm using the type of intermediate tables generated by the application of one operation [9]. To produce such type, the algorithm uses a forward subroutine that begins from the type of the input table and a backwards substituting that begins from the output table type [9]. In these routines, the algorithm applies a set of rules for forward and backward analysis to infer the type of the product or the input to an operator [9].

3.3.4 Implementation and Findings. The authors implemented Calico in Python and used the Pandas library for table transformation and the Vega-Lite library for the visualization [9].

To test the efficacy of Calico, the authors performed an experimental evaluation to answer three research questions [9]:

1. "Can Calico solve more visualization tasks than state-of-the-art approaches within a given time limit?" [9]
2. "Does Calico improve task-solving time compared to state-of-the-art approaches?" [9]
3. "How important are the individual refinement type rules for forward and backward analysis?" [9]

The evaluation consists of running Calico against a state-of-the-art solution, in this case Viser, on a set of 40 benchmarks obtained from different sources [9]. Unlike Calico, Viser does not posses any refinement types nor bidirectional analysis capabilities. To provide a baseline approach, an LLM was used to produce visualization program for comparison [9].

Out of the 40 benchmarks, Calico was able to find the solution for 39 of them, while Viser only solved 29, which was only slightly better than the LLM, which solved only 27 [9]. From these results the paper concludes that Calico is more effective at solving problems than Viser. Another metric measured by the authors is the time taken to find a proper solution. The cumulative time taken by Calico to solve all the benchmarks was relatively linear, while Viser's cumulative time grew exponentially with each additional benchmark [9]. The authors found that, on average, Calico takes 1.56 seconds to perform a visualization task, while Viser took 78.55 seconds, a 50x improvement [9]. Thus, they found that Calico is more scalable and bring a significant improvement of speed over Viser [9]. Finally, they tested Calico with a different version of Calico, called Calico-NR, which does not implement the related-operator predicate, to test the effectiveness of this predicate [9]. The results of this test showed that while the standard version of Calico can solve 39 benchmarks, Calico-NR solved 3% less benchmarks and required 22% more time [9]. Thus, the authors found that the related-operator predicate improves the performance of Calico [9].

The paper also discusses a simple usability study composed of an unclear number of users with basic knowledge in data analytics [9]. They asked the user to rate the usage of Calico from 1 to 5, where 1 is easy to use and 5 is hard to use [9]. The average score given by the participants was 1.8, which signify that Calico's notation is helpful and straight forward despite the learning curve [9].

3.3.5 Paper's Conclusion. The paper concludes with a discussion on related work on example-based program synthesis, refinement types, and automated visualization. It then provides a summary of the solution approaches and the findings of this paper.

4 Conclusion

Our paper surveys the modern applications of refinement types across three impactful domains: cybersecurity, robotics, and data visualization, through three recent academic papers.

In cybersecurity, we analyzed and explored how refinement types can be utilized at a binary level to detect cache-based side-channel vulnerabilities in cryptographic software effectively with the tool CaType. CaType performed overwhelmingly well in detecting both new and old vulnerabilities while outperforming other tools in efficacy and modeling defense strategies such as blinding. In robotics, we explored MARVeLus, which is a synchronous programming language for cyber-physical systems that utilizes refinement types. MARVeLus allowed for the effective and direct deployment of verified code into physical robots without a large translation overhead. Lastly, in data visualization, Calico, a domain-specific language, uses refinement types and bidirectional analysis to synthesize programs. Calico improves on existing data visualization tools by specifying output constraints using logical predicates, which in turn generates more accurate visualizations faster.

By studying each of these domains and their effective applications of refinement types, we can see the increasing relevance of refinement types as a solution for real-world problems. To conclude, it can be seen from our survey that as software systems continue to grow in complexity, expressive methods of verification such as refinement types will become increasingly important to maintain software that is usable, secure, and correct.

References

[1] Jiawei Chen, José Luiz Vargas de Mendonça, Shayan Jalili, Bereket Ayele, Bereket Ngussie Bekele, Zhemin Qu, Pranjal Sharma, Tigist Shiferaw, Yicheng Zhang, and Jean-Baptiste Jeannin. 2022. Synchronous Programming and Refinement Types in Robotics: From Verification to Implementation. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems* (Auckland, New Zealand) *(FTSCS 2022)*. Association for Computing Machinery, New York, NY, USA, 68–79. doi:10.1145/3563822.3568015

[2] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. *SIGPLAN Not.* 26, 6 (May 1991), 268–277. doi:10.1145/113446.113468

[3] Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. arXiv:2010.07763 [cs.PL] https://arxiv.org/abs/2010.07763

[4] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. 2022. Cache Refinement Type for Side-Channel Detection of Cryptographic Software. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1583–1597. doi:10.1145/3548606.3560672

[5] Fairouz D Kamareddine, Twan Laan, and Rob Nederpelt. 2004. *A modern perspective on type theory* (2004 ed.). Springer, New York, NY.

[6] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. doi:10.1145/2628136.2628161

[7] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. arXiv:1604.02480 [cs.PL] https://arxiv.org/abs/1604.02480

[8] Megha Wankhade and Suhasini Vijaykumar Kottur. 2020. Security Facets of Cyber Physical System. In *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*. 359–363. doi:10.1109/ICSSIT48917.2020.9214079

[9] Jingtao Xia, Junrui Liu, Nicholas Brown, Yanju Chen, and Yu Feng. 2024. Refinement Types for Visualization. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1871–1881. doi:10.1145/3691620.3695550