Static Analysis for Independent App Developers

Lucas Brutschy
Department of Computer Science
ETH Zurich
lucas.brutschy@inf.ethz.ch

Pietro Ferrara
IBM Thomas J. Watson
Research Center
pietroferrara@us.ibm.com

Peter Müller
Department of Computer Science
ETH Zurich
peter.mueller@inf.ethz.ch

Abstract

Mobile app markets have lowered the barrier to market entry for software producers. As a consequence, an increasing number of independent app developers offer their products, and recent platforms such as the MIT App Inventor and Microsoft's TouchDevelop enable even lay programmers to develop apps and distribute them in app markets.

A major challenge in this distribution model is to ensure the quality of apps. Besides the usual sources of software errors, mobile apps are susceptible to errors caused by the non-determinism of an event-based execution model, a volatile environment, diverse hardware, and others. Many of these errors are difficult to detect during testing, especially for independent app developers, who are not supported by test teams and elaborate test infrastructures.

To address this problem, we propose a static program analysis that captures the specifics of mobile apps and is efficient enough to provide feedback during the development process. Experiments involving 51,456 published TouchDevelop scripts show that our analysis analyzes 98% of the scripts in under a minute, and five seconds on average. Manual inspection of the analysis results for a selection of all scripts shows that most of the alarms are real errors.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages — Program analysis; D.2.4 [*Software Engineering*]: Software/Program Verification — Reliability; D.3.2 [*Programming Languages*]: Language Classifications — Specialized application languages

General Terms Languages, Reliability, Verification

Keywords Abstract Interpretation; Static Program Analysis; Mobile Applications; TouchDevelop

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660219

1. Introduction

Mobile app markets have transformed the software industry fundamentally by lowering the barrier to market entry for software producers. As a consequence, many apps are offered by independent developers rather than software companies. Recent platforms such as the MIT App Inventor [30] and Microsoft's TouchDevelop [29] enable even lay programmers and children to develop apps and distribute them in app markets.

A major challenge in this distribution model is to ensure the quality of apps. Besides the usual sources of errors, mobile apps are susceptible to a number of specific errors. For instance, the event-based execution model leads to error-prone non-determinism; mobile apps are executed in a volatile environment, where access to resources such as network connectivity may be intermittent and shared data such as the device's media collection may change arbitrarily; apps run on diverse hardware with different features such as sensors. These characteristics make mobile apps very difficult to test, especially for independent app developers, for whom it is typically not economically viable to have access to test teams and elaborate test infrastructures. The situation is even more difficult for lay programmers because they cannot be expected to be familiar with advanced testing techniques. When apps are developed directly on the device like in TouchDevelop, the limited resources preclude the use of test infrastructure such as hardware emulators. On the other hand, publishing a faulty app is potentially very harmful for a developer in a market that relies heavily on customer reviews.

To address this problem, we propose a static program analysis that allows app developers to detect errors in mobile apps. A static analysis checks a program for any sequence of input events, any execution environment, and any hardware configuration. Therefore, it reveals errors that are difficult to find during testing. Our analysis targets independent app developers by being fully automatic, by being sufficiently efficient to provide timely feedback during the development, and by favoring precision over scalability to minimize the number of spurious alarms. The latter choice is motivated by the fact the independent app developers tend to build apps of limited size such as small utilities and simple games. Our experimental results on 51,456 published scripts demonstrate

that our analysis analyzes 98% of the scripts in under a minute, and five seconds on average. Manual inspection of the analysis results for a selection of all scripts shows that most of the alarms are real errors.

We developed our analysis for TouchDevelop in order to focus on the specifics of mobile apps, which are largely orthogonal to the problems of analyzing programs written in general-purpose programming languages. The higher abstraction level of TouchDevelop programs (called *scripts*) allows us to define an analysis that is both precise and efficient. Moreover, many TouchDevelop scripts are written by laymen and on the device, which makes a static analysis particularly useful. TouchDevelop has become popular with over 90,290 scripts on the TouchDevelop marketplace [1]. The source code of all published scripts is publicly available and, thus, accessible to our analysis.

Most of the challenges our analysis deals with (such as an event-based execution model and a volatile environment) are not specific to TouchDevelop, but shared by other mobile platforms such as Android and iOS. These platforms pose several additional challenges such as a more involved event model, the need to analyze mainstream programming languages, a larger API, and more complex heap data structures. Nevertheless, we believe that our solutions could also be useful in the context of a more elaborate analysis for mainstream platforms.

TouchDevelop. TouchDevelop is a sequential, statically-typed, imperative programming language. Data is stored in local variables, persistent global variables, heap-allocated records and objects, and in cloud tables. The language and its libraries offer many features that simplify the development of mobile apps such as direct access to the mobile device (to its sensors, text-to-speech system, media library, etc.), asynchronous events to handle various forms of input, persistent variables, and support for web services (for instance, for handling XML and JSON structures). TouchDevelop scripts can be written and executed via a designated app or in any browser. They can also be bundled with a runtime system and offered as regular apps in the WindowsPhone Marketplace.

The following "Song Shaker" script, which is taken from a TouchDevelop tutorial, illustrates how TouchDevelop scripts interact with the mobile device. It declares an event handler that gets invoked each time the user shakes their device. The event handler accesses the media library via the API service **media**, selects a random element from the song list, and plays it.

```
event shake() {
  media→songs→random→play;
}
```

The script also illustrates a typical error in mobile apps: if the media library does not contain any songs when the phone is shaken, the script crashes. Testing will not reveal this error unless the developer has access to a configuration (on an actual device or in an emulator) with an empty song list. This is particularly unlikely when the script is being written on the device because the developer would have to delete all songs in order to catch the error. In contrast, the error is easily detected by a static analysis that has a semantics of the API operations and a suitable abstraction of collections.

The TouchDevelop platform is closely integrated with the cloud. In particular, every script that is being developed gets replicated in the cloud every 10 seconds. We build on this infrastructure to run our static analysis as a cloud service, which will allow us to provide timely feedback to developers, even if they write their scripts on devices that are not powerful enough to perform a static analysis.

Contributions and Outline. The main contribution of this paper is a precise and efficient static program analysis for the analysis of mobile apps, in particular, apps developed by independent app developers and lay programmers. To achieve that, we extend a generic abstract interpreter (Sec. 2) to address five key challenges in the static analysis of mobile apps:

1. How to detect and report errors in a late-failing model, where the execution may continue long after an error has occurred (Sec. 3).
2. How to obtain sufficiently precise information for collections without using an expensive heap analysis (Sec. 4).
3. How to abstract the execution environment of a mobile device precisely and efficiently (Sec. 5).
4. How to soundly abstract persistent variables, especially when apps are frequently aborted (Sec. 6).
5. How to improve the performance of the analyzer by reducing the abstraction of the execution environment to aspects that are actually relevant for the script to be analyzed (Sec. 7).

The focus of this paper is on providing (not necessarily novel) solutions to these challenges that strike a good trade-off between precision, efficiency, and soundness and, in combination, result in a practically-useful analysis. We briefly discuss our implementation and report experimental results in Sec. 8. We discuss related work in Sec. 9 and conclude in Sec. 10.

2. Generic Analyzer

We phrase our analysis as an abstract interpretation [9, 10]. In this section, we introduce the overall structure of the analysis; the individual components will be refined in the subsequent sections.

The structure of the abstract domain is formalized in Fig. 2.1. An abstract state in $\Sigma^\sharp$ consists of an abstract heap in $Heap^\sharp$ to track references and an abstract value state in $Value^\sharp$ to track information about primitive values such as numbers.

Abstract Identifiers. Both the heap and the value domains give values to *abstract identifiers in* $Id^\sharp$. An abstract identifier

$$
\begin{aligned}
\Sigma^\sharp &:= \mathit{Value}^\sharp \times \mathit{Heap}^\sharp \\
\mathit{Heap}^\sharp &:= \mathit{Id}^\sharp \to \mathit{Obj}^\sharp \\
\mathit{Value}^\sharp &:= \mathit{Numeric}^\sharp \times \mathit{String}^\sharp \\
\mathit{Id}^\sharp &:= \mathit{Var} \cup \left(\mathit{Obj}^\sharp \times \mathit{FieldNames}\right) \\
\mathit{Obj}^\sharp &\subseteq \mathit{Label}
\end{aligned}
$$

Figure 2.1: The structure of the abstract domains. The individual components will be refined in subsequent sections.

represents data that the analysis needs to track, in particular, the values of local variables (in *Var*) and of abstract heap locations (in $\mathit{Obj}^\sharp \times \mathit{FieldNames}$). Each abstract heap location represents a potentially unbounded set of concrete locations. Ferrara [15] showed how to combine the heap and value analyses in the presence of such identifiers. We will introduce more identifiers later to represent collections and components of the mobile device.

Heap Analysis. We adopt an allocation site-based abstraction of the heap. All objects allocated at the same program point (in *Label*) are summarized into one node, making the heap representation bounded. Since complex (in particular, recursive) data structures are uncommon in TouchDevelop, this abstraction yields an efficient yet sufficiently precise analysis. In Sec. 4, we will refine the heap abstraction by adding an abstract domain for collections.

Value Analysis. Our value analysis uses a combination of domains that approximate primitive values. An important application of the numerical domain $\mathit{Numeric}^\sharp$ in our setting is to check accesses to numerically-indexed collections. Therefore, we use Octagons [22] to efficiently track linear inequalities among abstract identifiers, but our implementation also supports other numerical domains. For relational domains, we treat summary identifiers following the technique introduced by Gopan et al. [16].

Besides numbers, our value analysis tracks string values, which are used in TouchDevelop for instance to encode enumerations and to access XML and JSON structures. Our string domain $\mathit{String}^\sharp$ assigns to each string variable a k-bounded set of string constants (or $\top$). In Sec. 3, we will introduce an additional value domain.

Abstract Semantics. A TouchDevelop script $c \in \Pi$ consists of a set A_c of actions (procedures and methods) and a set E_c of event handlers. Each action and each event handler has a statement as body. Statements (in *Stmt*) include the usual operations such as assignments, conditionals, loops, and action calls.

Following the abstract interpretation framework, we define a computable semantics of statements as a small-step abstract semantics $\widehat{\mathbb{S}} : \mathit{Stmt} \to \left(\Sigma^\sharp \to \left(\mathit{Label} \to \Sigma^\sharp\right)\right)$. For a given statement and initial abstract state, $\widehat{\mathbb{S}}$ yields a mapping from each program point (in *Label*) in this statement to the abstract state at that point. We call such a mapping a *state mapping*. The definition of $\widehat{\mathbb{S}}$ is standard. In particular, the semantics of loops and recursive calls is determined by a fixpoint computation. Calls are analyzed inter-procedurally. We refer the interested reader to Cousot [8] for more details.

We also use a function $\widehat{\mathbb{S}}_{\text{SCRIPT}} : \Pi \to \left(\mathit{Label} \to \Sigma^\sharp\right)$ that defines the semantics of an entire script. We assume here that program points are globally unique. This function, as well as the initial state of a script, will be defined in Sec. 6.

3. Challenge #1: Late Failing

Late failing is a common strategy to improve the robustness of applications. Instead of throwing an exception or aborting the execution when an error occurs, the erroneous operation results in a special value such as JavaScript's undefined or NaN (not a number), and the execution continues. Late failing is also common in programming models for mobile apps. For instance, while Objective-C provides exceptions, Cocoa, the library for programming iPhone applications, makes extensive use of the NSError object, which provides a late-failing mechanism.

TouchDevelop supports late failing through a special *invalid*-value $\mathit{Invalid}_T$ for every type T, including primitive types (we omit the subscript in the following). Whenever an operation cannot be completed successfully, it yields the invalid-value of the appropriate type. Examples include accessing a sensor that does not exist, accessing a collection out of bounds, or reading an uninitialized variable. An invalid-value may be used in only two contexts: as argument to the is invalid action and as right-hand side of an assignment. However, the execution of a script is aborted when an invalid-value is passed to a local action or an API action, or if it is used as condition in control structures such as conditional statements or loops.

Late failing often makes debugging difficult because the error may become visible long after it has occurred. Elaborate debuggers that help finding the cause of an error through conditional data break points or reverse debugging may not be available to independent app developers and lay programmers, especially when they develop scripts on the device. Therefore, the challenge for our static analysis is to provide error messages that point programmers to the cause of the error.

Example. The code snippet in Fig. 3.1 contains a potential off-by-one error. $boards is a list of Board objects. If $index is equal to $boards→size when accessing this list in line 2, the call to at will return *Invalid*. When post to wall is called on this value in line 6, the script crashes.

Solution. To provide useful error messages in the presence of late failing, our analysis reports not only where invalid-values may abort the execution, but also where they originate from. To achieve that, we extend the value domain from

```
1  if ($index ≥ 0 and $index ≤ $boards → size) {
2    $board := $boards → at ($index);
3  } else {
4    $board := media → create board;
5  }
6  $board → post to wall;
```

Figure 3.1: An example illustrating late failing. Line 2 contains an off-by-one error, which may cause the script to abort in line 6.

Sec. 2 with a designated *invalid-domain*. This domain tracks whether a value is valid or invalid, and for invalid-values where they may have been computed. Therefore, the domain maps each abstract identifier to a set whose possible elements are *Valid* and pairs of *Invalid* together with the program point where the invalid-value was created. The lattice structure is given by set operators (for instance, the join operator corresponds to set union):

$$InvalidDom^\sharp := Id^\sharp \to \mathscr{P}(\{Valid\} \cup (\{Invalid\} \times Label))$$

Example Revisited. In the example from Fig. 3.1, Octagons infer that $0 \leq \$index \leq \$boards \to size$ in the then-branch (line 2). Consequently, the abstract semantics of the assignment in line 2 determines that the right-hand side may result in a valid or in an invalid value and, thus, assigns $\{Valid, (Invalid, l2)\}$ to $board, where $l2$ is the program point of the call to at.

Since line 4 results in a valid value for $board, the join-operation after the conditional statement results in the set $\{Valid, (Invalid, l2)\}$ for $board. Since this set contains invalid-values, the check at line 6 that the argument to post to wall is valid fails. Tracking the origin of invalid-values allows us to report a precise alarm: "When calling post to wall in line 6, the receiver expression $board may contain an invalid value due to the call in line 2".

Discussion. Our abstract domain for tracking validity is efficient, yet sufficiently precise for most cases. The precision of the analysis could be improved further by tracking relational information. For instance, initialization code typically assigns valid values to a set of variables. Checking whether one of them contains a valid value is then sufficient to determine that they all do, but our non-relational domain is not able to conclude that.

Moreover, our approach does not always point directly at the cause of an error. For instance, arguably the error in Fig. 3.1 is caused by a wrong condition in line 1. Our experience so far suggests that the information where the invalid-value was created is sufficient to find the cause of an error, but one could also apply backward analyses for this purpose [26].

4. Challenge #2: Collections

Mobile apps typically make heavy use of collections, for instance, the song collection of the media library, the JSON objects parsed from a web query, or the sprites currently visualized on a game board. However, testing apps for different contents of collections is difficult. For instance, developers would need to change the data on their devices or have access to emulators in order to test for different song lists, would need stub implementations to test for different results of web services, and would need advanced GUI testing to create different sprite configurations on a game board. None of these can be expected from independent developers and lay programmers. The challenge for our static analysis is to provide a precise abstraction of collections without resorting to an expensive heap analysis, which is typically not needed for TouchDevelop scripts.

Example. The following code snippet obtains a JSON object, in this case, an object that maps strings (representing field names) to strings, and stores it in variable $presidents.

```
1  $presidents := web → download json(...);
2  if ( $presidents → keys → contains ("USA") )
3    $presidents → field ("USA") → post to wall;
```

To show that the access to the field *"USA"* in line 3 yields a valid value and, thus, that the subsequent post to wall does not crash, the analysis requires information about the contents of the $presidents map. In particular, the analysis must be able to infer from line 2 that there is an element in the map whose key is *"USA"*.

This example illustrates the need for a *must-analysis* that tracks which elements are *definitely* contained in a collection. Other examples require the analysis to show that certain elements are not in a collection. That is, they require a *may-analysis* that tracks which elements are *possibly* contained in a collection.

Solution. To track contents of collections, we complement the heap domain from Sec. 2 with a collections domain. In this domain, we treat different kinds of collections (maps, lists, and sets) uniformly as maps. Lists are represented as maps from natural numbers (indices) to the list elements. In TouchDevelop, the elements of a set are implicitly ordered, which allows us to use the same representation for sets.

Similarly to the treatment of objects in our heap analysis (Sec. 2), we represent a collection by the program point (label) at which it is accessed for the first time. To ensure that we need to track only a bounded number of keys and values, we also represent those via the program point where they are accessed or added. That is, we introduce additional abstract identifiers from $Label \times \{key\}$ and $Label \times \{value\}$ to represent the keys and values that are accessed at the given program point. Since a program point may be executed several times, these identifiers may represent many concrete keys or values. Additionally, for every collection $l \in Label$, we introduce another abstract identifier l_{length} to represent

the number of elements in the collection. The values of these abstract identifiers are tracked by our value and heap domains.

Our collections domain maps a collection (represented by a label) to a set of may-elements and a set of must-elements:

$$\underbrace{Label}_{collection} \to \underbrace{\mathscr{P}(Label)}_{may\ elements} \times \underbrace{\mathscr{P}(Label)}_{must\ elements}$$

The *may-elements* over-approximate the set of elements contained in a collections, while the *must-elements* under-approximate it. The join operator is therefore defined as $(a,b) \sqcup (c,d) := (a \cup c, b \cap d)$.

Example Revisited. Our analysis determines the safety of the access in line 3 of the above example as follows. The call to download json in line 1 introduces a new collection, represented by label $l1$. Initially, nothing is known about the contents and length of this collection. So the numerical domain tracks that $0 \leq l1_{length}$, while the collections domain maps $l1$ to $(\top, \emptyset)$, expressing that any element is possibly and no element is definitely contained.

The conditional statement in line 2 accesses the collection and, therefore, adds the abstract identifiers $(l2, key)$ and $(l2, value)$ to the abstract domain, where $l2$ is the label for line 2. Our string domain tracks that $(l2, key)$ has value *"USA"*, and the invalid-domain tracks that both identifiers are valid (TouchDevelop does not allow invalid-values in collections). Moreover, the collections domain is updated to map $l1$ to $(\top, \{l2\})$, gaining the knowledge that the collection must contain a key-value pair with identifiers $(l2, key)$ and $(l2, value)$. Using this information, our analysis determines that the collection access in line 3 yields a valid element, and therefore the call to post to wall is safe.

Discussion. When developing the collections domain, we built on experimental results for a large set of TouchDevelop scripts [4] to determine the appropriate trade-off between precision and efficiency. These experimental results show that, compared to a smashing analysis (where all elements are represented by a single abstract identifier that is weakly updated), the may-/must-analysis with a separate key-value pair for each program point reduces the number of alarms by 13.3%, while the average analysis time per script is increased from 0.62s to 0.82s. We conclude that the gain in precision outweighs the loss in performance.

When applying our collections domain to existing scripts, we observed that the must-analysis is indispensable for the keys of a map, but is hardly used for its values. Therefore, the must-analysis is essential for collections with non-numerical keys. For collections with numerical keys, it would be sufficient to track the length of the collection; Octagons are then usually able to show that the key is present, that is, non-negative and less than the length.

```
1  event shake {
2    if (media → songs → is empty → not)
3      media → songs → random → play;
4  }
```

Figure 5.1: An example illustrating changes to the environment. A sound analysis would report an alarm for line 3 because the song list could have changed between lines 2 and 3.

5. Challenge #3: Mobile Environment

Like other mobile apps, TouchDevelop scripts interact with the environment of the mobile device in three ways. First, they use API actions to perform operations on the device, for instance, to call a phone number or to store a picture in the media library. Second, they use API actions to query data from the device, for instance, to take a picture or to obtain a song from the media library. Third, they declare event handlers that get invoked by the environment when certain event occur, for instance, when the device is shaken. TouchDevelop provides over 1,000 API actions and supports 28 kinds of events.

It is impossible for independent app developers and lay programmers to adequately test the interaction of a script with the mobile environment. They are neither able to explore the possible interactions among event handlers nor to test their app for more than a few device configurations. The challenge for our static analysis is to reflect the non-determinism of the event-based execution and to provide a useful semantics to the large number of API actions. In particular, this semantics needs to reflect when aspects of the mobile environment may change.

Example. The event handler in Fig. 5.1 is a slight variation of the one described in the introduction. When it is invoked, it first checks if the song list in the media library is empty and, if not, plays a random song.

Even though the conditional statement guards the access to the song list, this event handler may crash because TouchDevelop provides no atomicity guarantees for accesses to the device. In particular, the device's media library may be modified by other (non-TouchDevelop) apps running on the mobile device between lines 2 and 3 such that songs may be non-empty in line 2, but empty in line 3.

Modeling this semantics soundly would lead to a large number of alarms for errors that rarely occur in practice. Even though these are true alarms, programmers would find them annoying since TouchDevelop does not provide a way to fix the errors because it has no feature that ensures exclusive access to parts of the device. On the other hand, assuming that data on the device never changes would ignore many errors that are likely to occur in practice, especially for volatile data sources such as the compass.

Solution. We present our approach to modeling the interactions with the environment in three steps. (1) We explain how we capture the interactions through events, (2) we discuss our general approach to defining the semantics of API actions, and (3) we explain how we model the mobile environment and changes to it.

Events. The execution of a TouchDevelop script starts by executing any of its actions[1]. When this action has terminated, the execution enters an event loop, which executes the handlers for incoming events sequentially. In particular, an incoming event does not interrupt the execution of actions or event handlers, that is, there is no interleaved execution. Therefore, a single execution of a TouchDevelop script c with actions A_c and event handlers E_c can be represented by a sequence $a \to e_1 \to ... \to e_n$ where $a \in A_c$ and $\forall i \in [1..n] : e_i \in E_c$. We capture this execution model in the semantics as follows: First, we analyze each action in A_c separately for an initial state σ_0 that contains only information that is true for all possible environments. Second, we join the states after these public actions to obtain the initial state of the event loop. Third, we analyze the event loop by computing a fixpoint of all event handlers in E_c in order to soundly handle any number and order of events. See below for a formalization of this semantics.

API Semantics. In contrast to the actions defined in a TouchDevelop script, we cannot obtain the semantics of API actions by analyzing their bodies because they are implemented natively and typically cannot even be expressed in TouchDevelop. Re-analyzing the API actions during our inter-procedural analysis would also not be efficient.

To obtain a static analysis that is both efficient and sufficiently precise, we define the semantics of each API action lazily. We start from the following imprecise semantics, which we extracted from the informal API documentation:

1. If the API action may return *Invalid*, we precisely define the conditions under which this happens. Otherwise, we define the return value to be $\top$ in the value domains (and *Valid* in the invalid-domain).
2. If the API action may have side-effects, we define an abstract semantics that applies an over-approximation of the documented side-effects to the abstract state.

The precise treatment of invalid-values is necessary to avoid false alarms whenever a script operates on the result of an API call. Otherwise, the above semantics yields a sound, efficient, but extremely imprecise analysis. Starting from this initial semantics, we improved the semantics of an API action whenever we found that it caused false alarms in existing TouchDevelop scripts.

This approach is not only important for a pragmatic implementation of the analysis, but also to preserve its efficiency. It ensures that only the relevant aspects of the API are represented in detail by our abstract semantics, while we adopt a rough and efficient representation for everything else.

Environment Representation. To represent the state of the environment that can be accessed through API actions, we introduce abstract identifiers in addition to those described in Secs. 2 and 4. These additional *environment identifiers* represent for instance the song list and picture albums of the media library and the values obtained from various sensors. The values of the environment identifiers are tracked by the abstract domains described so far. The environment identifiers are also used to define the semantics of the API actions that access the environment. For instance, the semantics of **media** $\to$ songs is defined to yield the value of the environment identifier representing the song list.

An important aspect of the semantics is what we assume about the stability of the environment. As we have illustrated above, modeling the actual behavior, where the environment may change arbitrarily between any two operations, is impractical, whereas assuming that the environment is stable is highly unsound. To address this issue, we partition the set EId of environment identifiers into three sets: (1) Identifiers in $EId_{volatile}$ represent parts of the environment that change very frequently, for instance the values of certain sensors such as the compass. (2) Identifiers in EId_{stable} represent parts of the environment that are statically unknown, but stable during the execution of a script, for instance, the existence of a front camera. (3) Identifiers in $EId_{occasional}$ represent parts of the environment that in theory may change during the execution of a script, but will rarely do so, for instance the device's song list.

We capture these behaviors in our semantics as follows. For the environment identifiers in $EId_{volatile}$, we track only information that holds in all possible states. Since these parts of the environment are volatile, any other assumptions about these identifiers would make it likely for the analysis to miss errors. The information about the environment identifiers in EId_{stable} is persistent throughout the execution of the script. For example, once the script has checked that a front camera exists, this information will be retained for the analysis of the rest of the script. For the environment identifiers in $EId_{occasional}$, we use the following compromise between soundness and practicality. We assume them to be stable throughout the execution of one action or event handler, but allow them to be modified in between. This semantics is practical because it avoids alarms for code like in Fig. 5.1, and it is unlikely to miss errors that actually occur in practice because most actions and event handlers terminate quickly, which makes it unlikely that the environment changes during their short execution.

Formalization. We define the semantics of executing one action or event handler as follows:

[1] We ignore the difference between public and private actions here for simplicity.

$$\textsc{Action}_c(\sigma) \quad := \quad \lambda l.\ \bigsqcup_{a \in A_c} \widehat{\mathbb{S}}[\![a]\!](\sigma)(l)$$

$$\textsc{Event}_c(\tau) \quad := \quad \lambda l.\ \bigsqcup_{e \in E_c} \widehat{\mathbb{S}}[\![e]\!]\left(\bigsqcup_{l' \in Exit_c} \tau(l')|_{\overline{EId_{occasional}}}\right)(l)$$

For a script c, $\textsc{Action}_c$ takes an initial abstract state σ and yields the state mapping that is obtained by executing any action of the script. For a label l, this mapping yields the result of applying the semantic function for the body of an action a to l (see Sec. 2) and then applying the join-operator over all actions in the script.

$\textsc{Event}_c$ is defined similarly, but takes a state mapping τ as argument rather than just a state. It applies the semantic function for the body of an event e to the state obtained by joining the states at all exit labels l' in τ. An *exit label* is the program point after the last statement in an action or event handler; $Exit_c$ denotes the set of all exit labels in script c. Before computing this join, we remove all information about the environment identifiers in $EId_{occasional}$ from the exit states because these identifiers may change arbitrarily after each action or event handler. This is achieved by the operator $|_{\overline{EId_{occasional}}}$, which takes an abstract state and, for each environment identifier in $EId_{occasional}$, assigns $\top$ in the value domains and, depending on the part of the environment an abstract identifier represents, *Valid* or $\top$ in the invalid-domain.

Finally, the semantic function $\widehat{\mathbb{S}}_{\text{EV}}$ takes a script and an initial abstract state, and yields a state mapping. It computes the fixpoint for the event loop. The ∇ operator computes the join of all incoming state mappings and, after a fixed number of iterations, instead applies widening to accelerate and ensure the convergence of the analysis.

$$\widehat{\mathbb{S}}_{\text{EV}}(c, \sigma) \quad := \quad lfp^{\sqsubseteq}_{\textsc{Action}_c(\sigma)} \lambda\tau.\,(\tau \nabla \textsc{Event}_c(\tau))$$

Example Revisited. Assume that the abstract identifier $mss \in EId_{occasional}$ represents the number of songs in the media library. In the abstract state after line 2 of Fig. 5.1, we have $mss > 0$. Our abstract semantics retains this information throughout the body of the event handler, which allows it to conclude that random yields a valid value.

For comparison, consider the following script, which checks only once at the beginning of the script if the song library is empty and, if so, terminates the event loop:

```
1  action main {
2    if ( media → songs → is empty)
3      time → stop;
4  }
5
6  event shake {
7    media → songs → random → play;
8  }
```

Like for Fig. 5.1, our analysis infers $mss > 0$ at the end of main. However, this information is dropped from the exit state of main before applying the semantic function for shake. As a result, the analysis issues an alarm on line 7, stating that the result of random may be invalid and that play cannot be executed on an invalid-value.

6. Challenge #4: Persistent Storage

Instead of using a traditional file system, some mobile app platforms provide a more abstract way of storing objects persistently in order to simplify programming and to avoid security issues. For instance, Android's SharedPreferences store key-value pairs persistently. TouchDevelop stores the values of all global variables, and everything reachable from them, persistently. The runtime environment automatically loads persistent data when the script is started. Every update to a global variable (or an object reachable from them) is immediately reflected in the persistent storage. In the following, the term *persistent variable* subsumes the global variables of a script and all memory locations reachable from them.

Persistent variables may lead to two kinds of errors. First, a programmer might break the initialization code for a persistent variable *after* the variable has been initialized on their device (during an earlier run of the script). When the script with the broken initialization code is executed on other devices, variables will remain un-initialized. This problem is found during testing only if the tester explicitly resets the persistent storage on their device.

Second, some mobile platforms may abort apps without notifying them. For instance, iOS may suspend an app and may purge a suspended app when the system runs out of memory. Similarly, a TouchDevelop script may get aborted at any time. When the script is re-started later, its persistent variables will have the values they had when the script was aborted. These values may be unexpected, for instance, because they do not satisfy invariants that the programmer intended to maintain. This kind of error is difficult to test since one would have to abort the script after each update of a persistent variable and observe whether it behaves correctly after a re-start. The challenge for our static analysis is to soundly model persistent storage such that it can detect errors caused by faulty initialization code and by abortion.

Example. The script in Fig. 6.1 declares a persistent global variable level in line 1. Let's assume that $boards is a list with exactly three elements. To ensure that the access in line 4 yields a valid result, the programmer intends to maintain an invariant that $0 \leq$ **data** $\rightarrow$ level ≤ 2 at line 4. However, this invariant may be violated if line 6 sets **data** $\rightarrow$ level to 3, and the script gets aborted before line 9. When the script is re-started, **data** $\rightarrow$ level is 3, and the script crashes in line 4. From now on, the script will always crash until the user resets the persistent storage manually.

```
1  data level : Number
2  action main {
3    ...
4    $boards → at(data → level) → post to wall;
5    ...
6    data→level := data → level +1;
7    if (data → level = 3) {
8      "Game over!" → post to wall;
9      data → level := 0;
10   }
11 }
```

Figure 6.1: A script illustrating the use of persistent variables. The developer intended to maintain an invariant at line 4 that the global variable **data**→level is at most two. This invariant may be violated if the script gets aborted between lines 7 and 9.

Solution. To detect the kinds of errors described above, our analysis models that a script could be started either in a fresh state (if it is the very first execution on the device or if the persistent storage has been reset since the last execution) or in a state that resulted from terminating the previous execution at any point (by aborting it or by executing it to completion). That is, we view the process of starting, aborting, and re-starting a script as an iteration and analyze its effect on the persistent variables. Technically, our analysis computes two nested fixpoints. The inner fixpoint handles the event loop of a single script as was explained in the previous section. The outer fixpoint handles re-starts of the entire script as is explained in the following.

A fresh execution starts in an initial abstract state σ_0 with the following properties: (1) Global variables are set to the default values of their types T, that is, 0 if T is Number, the empty string if T is String, and *Invalid* otherwise. (2) Abstract environment identifiers in $EId_{stable} \cup EId_{occasional}$ are set to $\top$ in the String and numerical domains. The state of the invalid-domain reflects whether the TouchDevelop platform guarantees that their values are definitely valid (for instance, for **media** → pictures) or not (for instance, for **senses** → front camera). We extend the initial abstract state σ_0 to an initial state mapping τ_0, which maps the start label of each action to σ_0, and all other labels to $\bot$.

The function SCRIPT_c computes a state mapping that reflects one execution of the entire script c, starting from an initial state mapping τ, which is either the initial state mapping or a state mapping obtained from a previous execution. $\widehat{\mathbb{S}}_{\text{EV}}$ denotes the computation of the inner fixpoint, see Sec. 5.

$$\text{SCRIPT}_c(\tau) \quad := \quad \widehat{\mathbb{S}}_{\text{EV}}\left(c, \bigsqcup_{l \in Label_c} \tau(l)|_{P_c \cup EId_{stable}}\right)$$

To reflect that the previous execution might have been aborted at any program point, we determine the start state for the next execution by joining the states at all labels of the script c. If τ is the initial state mapping τ_0, this join yields the initial state σ_0. Before computing the join, we apply the operator $|_{P_c \cup EId_{stable}}$ to remove all information from the abstract states except the values of the abstract identifiers P_c for the persistent variables and the values of the abstract environment identifiers EId_{stable} for the stable parts of the environment. To remove information for an abstract identifier, the operator assigns $\top$ in the value domains and, depending on what the identifier represents, *Valid* or $\top$ in the invalid-domain.

We can now define the semantic function $\widehat{\mathbb{S}}_{\text{SCRIPT}}$ that we introduced in Sec. 2. It takes a script and yields a state mapping by computing the fixpoint for the abort-restart loop, that is, the outer fixpoint of our analysis. It applies widening to ensure termination:

$$\widehat{\mathbb{S}}_{\text{SCRIPT}}(c) \quad := \quad lfp^{\sqsubseteq}_{\tau_0} \lambda\tau.(\tau \nabla \text{SCRIPT}_c(\tau))$$

Example Revisited. Applying our analysis to the script in Fig. 6.1 detects the error described above. In the initial state σ_0, **data** → level has its default value 0. After the first execution of the script, that is, in the state mapping $\text{SCRIPT}_c(\tau_0)$, **data** → level is 0 in line 4, and 1 after line 6. Therefore, the next application of SCRIPT_c in the outer fixpoint computation uses the interval $[0,1]$ as the abstract value of **data** → level in line 4. Depending on the widening limit, the analysis infers that after some iterations, we have **data** → level $\in [0,3]$ or just $0 \leq$ **data** → level. In either case, the analysis reports an alarm that the access to $boards in line 4 may yield an invalid-value such that the subsequent post to wall might crash.

Discussion. Our experience shows that the computation of the outer fixpoint over all script executions is expensive, but necessary in practice. To increase efficiency, we considered two alternatives.

First, one could avoid the outer fixpoint computation by not making any assumptions about the persistent variables at the beginning of an execution. However, this solution causes false alarms for two major reasons. (1) It loses information about the initialization of persistent variables from default values or previous script executions. Therefore, the analysis must consider the option that all persistent variables may initially have invalid values, which leads to many false alarms. For instance in Fig. 6.1, the analysis would consider the case that **data** → level may be initially invalid, such that the call to at in line 4 would crash the script. (2) Making no assumptions about the persistent variables at the beginning of an execution means in particular that the heap analysis would have to consider maximal aliasing among heap structures such that all updates to global data would have to be non-destructive. To evaluate this alternative, we performed an experiment on 334 randomly selected scripts. Avoiding the outer fixpoint computation reduced the average analysis time per script from

1.03 to 0.71s. However, the number of alarms increased from 336 to 378. That is, the outer fixpoint computation avoided 42 false alarms; almost all of them are caused by the two reasons mentioned above. We decided that avoiding these false alarms justifies the slower analysis.

Second, one could speed up the computation of the outer fixpoint by unsoundly ignoring script abortions and considering only the exit state of terminating scripts. Experiments on the same set of scripts showed that, compared to our solution, this approach reduces the average analysis time per script from 1.03 to 0.63s and the number of alarms from 336 to 333. All three alarms missed by the unsound alternative are real errors (in the scripts *appla* and *arwba*). We decided to adopt the sound treatment of abortions because the analysis is sufficiently efficient for almost all TouchDevelop scripts such that we can afford to be sound.

7. Challenge #5. Demand-Driven Abstraction

As we have discussed in Sec. 5, an analysis for mobile apps requires precise information about the execution environment of the mobile device. This environment typically contains a large number of components. For example, the TouchDevelop API makes 15 different components of the device accessible through **media** and **senses** alone. Many of these, such as the media library, are represented by nested object structures, which are reachable immediately after the start of the script. Moreover, some frequently used objects provide a large number of fields. For instance, instances of the **box** type, which represents user interface elements, provide 42 properties. Introducing abstract identifiers for all components of the mobile device and all fields of the used objects leads to an intractable amount of reachable abstract identifiers and, thus, an inefficient analysis. The challenge for our static analysis is to reduce the number of abstract identifiers without sacrificing precision.

Example. Even though the following script uses only two components (game boards and the front camera), a naïve analysis would create abstract identifiers for every reachable field in the initial state.

```
action main {
  var $board := media → create board(640);
  $board → create rectangle($board → width/2,
                            senses → front camera → height);
}
```

Moreover, the abstract semantics of **media** → create board will initialize all fields of the new board to their default values. However, most of these fields, such as the background color, a background image, and a reference to a Camera object, are irrelevant for the script above.

Solution. Before running the actual analysis of the script, we use a type-based static pre-analysis to collect which fields are actually accessed by the script. This pre-analysis yields an element in $\mathscr{P}(\mathit{Type} \times \mathit{FieldName})$, which is used by the actual analysis to determine which abstract identifiers are needed to represent the relevant parts of the execution environment and the relevant fields of objects. Since TouchDevelop is statically typed and has no subtyping, computing these fields is straightforward.

Example Revisited. For the script, our pre-analysis infers { (Board,width), (**senses**, front camera), (Camera,height) }. The initial state σ_0 will only represent those three fields and omit all other components of the environment.

Discussion. The type-based pre-analysis is a simple, yet effective way to increase the performance of the analysis. For the above script, it reduces the analysis time from 47.71 to 0.43s.

Instead of our coarse type-based pre-analysis, one could use a conservative pointer analysis, possibly in combination with a live variable analysis, to determine sets of objects instead of types for which certain fields should be represented. We have not explored this options since our experiments showed that our simple pre-analysis is sufficient.

8. Experimental Results

In this section, we present and discuss the experimental results of our analysis in terms of performance and precision. For these experiments, we configured our analysis to emit alarms when an invalid-value is passed as argument to an action (which include arithmetic operations). With late failing (see Sec. 3), this check covers many faults in the code, most importantly:

- Out-of-bounds access to collections
- Access to unavailable device features (e.g., a front camera)
- Access to unavailable resources (e.g., network connection)
- Operations on uninitialized persistent variables
- Operations on canceled or invalid user input
- Failed conversions (e.g., string to date)
- Operations on non-existent data (e.g., a contact with an invalid phone number field)

Besides these common errors, our analysis can report additional issues without extending the abstract domain. These include for instance drawing outside of picture bounds (which has no effect), invalid numeric computations such as division by zero (which produces a *NaN* value, which may be handled correctly by subsequent code), and violations of certain API protocols such as pausing a media player only in states in which it is not playing (which has no effect). These issues are not necessarily fatal errors, but may indicate undesired program behavior. Since it is often difficult to judge whether such issues are actual errors, we do not check them in the experimental evaluation.

8.1 Implementation

We implemented our analysis on top of Sample, a generic static analyzer that supports the combination of various heap

and value analyses [15]. We used the Octagons implementation of APRON [19] as numerical domain and an abstract domain of k-bounded sets of constant strings (with $k = 3$) as string domain.

We extended Sample with a standard inter-procedural analysis based on a supergraph approach, which corresponds to a control flow graph where each callee is connected by an edge to its call and return site. To make this simple approach scalable, we employed a restricted form of access-based localization of abstract states [24]. When we analyze an action call, we propagate directly to the return site of the call the portion of the abstract state that is not accessed inside the callee action (according to the type-based pre-analysis described in Sec. 7). For the portion that is accessed or modified inside the callee action, we rely on the abstract semantics of the action's body.

The analysis is accessible to TouchDevelop users through a preliminary web interface that can be found under the following URL: `http://tb.inf.ethz.ch` . In cooperation with the developers of TouchDevelop, the analysis described in this paper is currently being integrated in the programming environment itself.

8.2 Performance

For an evaluation of the performance of our analysis, we ran it on a wide set of existing TouchDevelop scripts. From all scripts published before March, 22^{nd} 2014, we selected all scripts that satisfy the following properties:

- The script is accessible through the TouchDevelop cloud API and has neither syntax nor type errors.
- The script is a root script, that is, a newly published script rather than a variation of an existing script. Focusing on root scripts avoids a bias towards popular scripts of which several tens of variations are published.

Out of 90,290 scripts published at the time of writing, 51,456 scripts satisfy these criteria. We were unable to analyze 335 scripts because they use very recent or even experimental and undocumented language features which our tool does not support yet. 96 scripts could not be analyzed due to a memory error in the most recent version of APRON, which has been confirmed but not yet fixed.

On the remaining 51,025 scripts, we ran our analysis with a time-out of one minute per script. 594 scripts (1.1%) could not be analyzed within this limit. Time-outs were mainly caused (1) by a large number of numerical variables, for which the cubic complexity of Octagons is too high, and (2) by dense call-graphs, which is a well-known problem for whole-program analyses. The first problem could be addressed by reducing the number of variables through a more aggressive pre-analysis (see Sec. 7) or by using a less precise numerical domain. For the second problem, we may adopt a less precise inter-procedural analysis. Nevertheless, we did not explore these optimizations since we believe that being able to analyze more than 98% of the scripts in less than a minute is sufficient to provide feedback to app developers.

The following evaluation is based on the 50,431 root scripts that were successfully analyzed within the one-minute limit. We will refer to this set of scripts as the *root set*. We assessed the run time and the number of alarms for the root set. A qualitative analysis of the alarms in presented in the next subsection.

We ran all the experiments on an Intel Core 2 Quad CPU (2.83GHz) and 4GB of memory running Ubuntu Linux 13.04. The analyzer is written in Scala (and therefore compiled to Java Bytecode), and we executed it on an Oracle Java 7 Virtual Machine.

Table 1 reports the experimental results obtained by our analysis. The first row summarizes the results for all scripts in the root set. The 50,431 scripts have in total 17MLOC (an average of 340 LOC per script)[2]. The analysis takes in total about 2 days and 22 hours, that is, on average five seconds per script. This shows that our analysis is efficient and may be applied during the development of a script, giving timely feedback to the developer in most cases.

TouchDevelop developers have the option to categorize their scripts using pre-defined tags. In Table 1, we also report the experimental results for the root scripts in some categories. We noticed that a relatively small number of root scripts is tagged, probably because root scripts are early versions of a script and developers tend to tag more mature versions. It is interesting to observe that, while the tagged scripts are on average shorter than the average root script, our analysis reports more alarms for the tagged scripts. We suspect that the tagged scripts perform more complex computations and interactions with the device than an average script. Therefore, they are likely to contain more errors but also to lead to more false alarms (like the CloudHopper script that we discuss in next subsection) in case our analyzer is not precise enough for these complex computations.

Comparing different categories shows that scripts in the games category are shorter than the average script, but are more expensive to analyze. We believe that games take longer to analyze because (1) they modify large portions of the state (for instance, a large number of sprites on the screen) during a recurring event loop, and (2) the placement of game elements on the screen is based on numerical computations that are expensive to handle in the numerical domain. It is also interesting to note that the average length of tagged scripts is less than the average length of all root scripts (and that the most frequently executed scripts, which are described in the

[2] TouchDevelop has no textual syntax since the editor works directly on the abstract syntax tree. Therefore, there is no fixed textual representation that can be used to count the number of code lines. We chose to count the number of lines in the textual representation returned by the TouchDevelop compiler API, which may differ from the visualization in the editor. Note that experiments have shown that Android implementations are about four times longer than TouchDevelop scripts with the same functionality [23].

	#Scripts	#LOC		#Alarms		Time	
		Sum	Avg.	Sum	Avg.	Sum	Avg. [s]
root set	50,431	17,149,776	340.06	23,466	0.47	2d22h29'04"	5.03
games	199	60,418	303.60	533	2.68	22'17"	6.71
entertainment	169	32,156	190.27	304	1.80	7'44"	2.74
tools	108	11,951	110.65	123	1.14	1'54"	1.06
music	65	4,584	70.52	146	2.25	2'11"	2.01
education	65	8,745	134.53	152	2.34	1'49"	1.67
productivity	51	6,757	132.49	147	2.88	2'05"	2.44
kids	49	13,298	271.38	58	1.18	4'40"	5.71
action	43	11,858	275.76	52	1.20	4'36"	6.42

Table 1: Results of the analysis for the root set and for some script tags. The columns show the number of scripts, code lines, and alarms, as well as the total processing time.

next subsection, are all shorter than the average of all root scripts).

8.3 Precision

To evaluate the precision of the analysis, we manually inspected the alarms for some scripts and checked how many of them are false alarms. Since it is not feasible to manually inspect the alarms for all scripts, we focus our inspection on two sets of scripts, one containing a random selection and the other one consisting of the ten most frequently executed scripts. We define the ratio $\frac{\#\ true\ alarms}{\#\ alarms}$ as the *precision* of the analysis.

Random Scripts. To evaluate the precision of our analysis on average scripts, we chose 51 scripts randomly by taking all those scripts from the root set whose randomly-assigned identifier starts with "aa". For these scripts, our analysis reported 34 alarms in total. Five of these are false alarms, which yields a precision of 85%. This precision rate is far above state-of-the-art static analyzers and suggests that our analysis may provide useful feedback to programmers without overwhelming them with false alarms.

Out of the 29 true alarms, one is caused by accessing an uninitialized persistent variable, ten are caused by incorrect assumptions about the media library (for instance, assuming that at least one song is in the library), eleven are caused by incorrect assumptions about the capabilities of the device (for instance, about the availability of sensors), one is caused by a missing check of user input, and six are caused by not checking whether the JSON result from a web service contains the expected fields.

Out of the five false alarms, three are in the script *aanpa*[3], which stores a collection persistently. Our heap analysis represents the content of this collection as summary node, such that the analysis cannot prove that the collection is non-empty. We could address this issue by adopting a more precise heap abstraction, but this would affect performance negatively. For the other two false alarms, our analysis abstracts away some disjunctive information about boolean variables. We could address this issue by adopting trace partitioning [21], which is already supported by Sample. However, trace partitioning needs to be tuned manually and would slow down the analysis.

Most Frequently Executed Scripts. To evaluate the precision of our analysis on scripts that users seem to find useful, we analyzed the ten most frequently executed scripts. (The number of executions is provided by the TouchDevelop platform.) These scripts are not necessarily in the root set, that is, they may be variations of other scripts. They range from 30 to 333 lines of code. The most frequently executed scripts include five games (doodle jump in two versions, CloudHopper, Line Runner, and BreakIt! Touch) and five small utilities. Such utilities are similarly popular on other platforms, see for instance the immensely popular flashlight apps on Google Play and Apple's App Store.

The analysis times for the utilities range from 0.01 to 1.47 seconds, whereas the times for games range from 4 to 20 seconds, confirming our earlier observations about games (see Table 1).

Among the five utilities, only My Online Meetings produced false alarms. They are all caused by code that concatenates and splits strings, and then makes assumptions about the size of the resulting strings. We could address this issue by adopting a more precise (and more expensive) string analysis [7]. However, our experience shows that this additional precision would be useful only in a few cases; therefore, we decided to opt for a faster but less precise string analysis. Among the five games, two produced false alarms. Nine out of the 14 false alarms that we found occur in one script (CloudHopper). All of them are caused by the fact that our analysis fails to infer the size of a collection because of missing narrowing operators.

With 30%, the precision obtained for the most frequently executed scripts is much lower than for the random scripts. This result can be explained in part (1) by the fact that these

[3] The source code of a script with identifier *PID* is available at `http://www.touchdevelop.com/api/PID/text`.

Script	PID	#LOC	#Runs	Time [s]	#True	#False
Flip a Virtual Coin!	htmh	30	45,300	0.01	0	0
My Online Meetings	mpuj	333	41,100	1.47	0	3
WiFi/3G Swap	kmjn	44	40,800	0.07	0	0
Line Runner	dvvx	261	16,600	11.48	1	0
internet speedtest	qwzu	39	9,150	0.05	0	0
doodle jump	ajkc	221	9,000	9.14	1	0
where am I ??	kblp	98	8,100	0.40	2	0
CloudHopper	wbxsa	255	8,050	19.43	1	9
BreakIt! Touch	zids	180	7,700	3.98	0	2
doodle jump	ybcy	221	6,700	6.10	1	0
Total		1,682		52.13	6	14

Table 2: Results of the analysis for the ten most frequently executed scripts. For each script, the columns show the identifier, number of code lines, number of executions, total processing time, as well as the number of true and false alarms.

scripts are very well tested during several thousand executions and, therefore, are likely to contain fewer errors, resulting in fewer true alarms (one per 280 code lines versus one per 175 code lines for the random scripts); and (2) by the CloudHopper script being on outlier. Discounting these two effects would increase the precision to 62%. Finding six true errors in these frequently executed scripts illustrates that our static analysis is able to detect errors that are not found during extensive testing.

9. Related Work

To the best of our knowledge, the only generic analyzer that has been applied to detect errors in mobile (Android) apps is Julia [25]. Julia extends a generic analyzer for Java to handle some typical features of Android apps such as representing user interfaces through XML files. Julia checks mostly for termination and null pointer accesses. It analyzes projects between 54 and 6300 LOC in the order of minutes, but with a very high rate of false alarms. Compared to Julia, our analysis benefits from the higher abstraction level of TouchDevelop code, such that the performance and precision numbers cannot be compared in a meaningful way.

Clousot [13] is a generic static analyzer for .NET that checks for run-time errors as well as user-provided assertions. Clousot relies on contracts to perform an intra-procedural analysis, while our analysis is inter-procedural. This makes our analysis less efficient, but avoids the need for annotations, which seems more suitable for independent app developers and lay programmers.

ASTRÉE [3] analyzes embedded software to prove the absence of run-time errors. Like our analysis, it combines various abstractions (numerical and heap domains), and handles the interaction with various external components (such as sensors). ASTRÉE is highly specialized to obtain zero false alarms for specific industrial software, while we targeted arbitrary TouchDevelop scripts.

Null pointer analyses [18] are similar to analyzing late failing. Like invalid-values, computing a null value is not an error, while certain uses (especially dereferencing) are. Our approach of tracking the origin of an invalid-value seems applicable to improve the feedback provided by null pointer analyses.

Heap analysis has been widely explored leading to various pointer [28] and shape [27] analyses with different trade-offs between efficiency and precision. Since TouchDevelop scripts mostly rely on built-in collections and objects rather than implementing their own heap structures, we adopted a standard allocation site-based pointer analysis and complemented it with a collections domain. This combination is efficient and sufficiently precise for typical TouchDevelop scripts.

Marron et al. [20] introduce an analysis to approximate the contents of sets. It extends shape analysis to track equality and subset relations among sets of references. These relations are must-relations, whereas our collections domain tracks must- and may-elements. Besides sets, our domain also supports lists and maps.

Dillig et al. [12] describe a very expressive analysis to determine may- and must-properties of the contents of containers (such as maps and sets). Their work adopts numerical first-order logic constraints to specify the elements of containers. Practically, their approach relies on an SMT solver that is called for each access to a collection. Our collection analysis tracks may- and must-information as well, but it is less precise than their approach. The design of our abstract domain has been driven by the properties we needed to track for existing TouchDevelop scripts, which are usually simple enough to not require full first-order logic expressiveness. The performance of our overall analysis is similar to theirs, although it includes a rich and precise model of the mobile environment and semantics, not just properties of containers.

Existing static analyses of array contents [11, 17] have a different focus than our collections analysis. On the one hand, they are less general because they focus on collections

of numbers, whereas we support arbitrary content, such as strings. On the other hand, for arrays of numbers, they can infer more complex invariants using expressive numerical domains.

Several recent analyses target security properties of mobile apps such as inference of Android permissions [14] or information flow [6]. They do not require a precise abstraction of the mobile environment, only knowledge on which APIs require permissions or read confidential information. They also do not have to capture the non-deterministic interaction among events since it is sufficient to know which events are reachable (to check the required permissions), or how information flows within an event handler. In contrast, our analysis crucially depends on a precise abstraction of the environment and on possible interactions among event handlers to detect errors.

TouchDevelop as well has been the target of an information flow [31] and a data flow [2] analysis to detect leaking of confidential data and cloned apps, respectively. These analyses do not need to precisely abstract the mobile environment and the event semantics.

10. Conclusions and Future Work

We presented a static analyzer for mobile apps written in TouchDevelop. In order to support independent app developers and lay programmers, we designed an analysis that does not require user input, is efficient, and produces a low number of false positives. These goals guided our solutions to five challenges of analyzing TouchDevelop code. Our experimental results demonstrate that our design indeed achieves these goals and, thus, a practically-useful analysis. In particular, our analysis detects errors that are very hard to test such as errors caused by certain sequences of events or abortions of the app. Our analysis benefits from the simplicity and high abstraction level of TouchDevelop. Although analyses for other mobile platforms such as Android and iOS face similar challenges and may be able to adopt some of our solutions, they will be more involved.

Besides improving the quality of apps, we also hope to achieve an educational effect, especially on lay programmers, by detecting errors that are easy to miss otherwise.

As future work, we plan to extend our analysis to some recent TouchDevelop features such as cloud types [5], which we currently handle imprecisely. In cooperation with the developers of TouchDevelop, the analysis described in this paper is currently being integrated in the programming environment itself. Interesting future work will be to evaluate how end-users interact with our analysis and to reflect these findings in the analysis design.

Acknowledgments. We are grateful to Nikolai Tillmann for many helpful discussions about TouchDevelop, to Antoine Miné for his help with the Apron library, and to Yves Bonjour for implementing the container analysis.

References

[1] TouchDevelop Web API cloud statistics. `https://www.touchdevelop.com/api/stats`. Accessed: 2013-09-17.

[2] M. Akhin, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal. Code similarity in TouchDevelop: Harnessing clones. Technical report, Microsoft Technical Report MSR-TR-2011-103, 2011.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of PLDI '03*. ACM Press, 2003.

[4] Y. Bonjour. Must analysis of collection elements. Master's thesis, ETH Zürich, 2013.

[5] S. Burckhardt, M. Fähndrich, D. Leijen, and B. Wood. Cloud types for eventual consistency. In *Proceedings of ECOOP '12*, LNCS. Springer, 2012.

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of MobiSys '11*. ACM, 2011.

[7] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *Proceedings of ICFEM '11*, LNCS. Springer, 2011.

[8] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. IOS Press, 1999.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*. ACM, 1977.

[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of POPL '79*. ACM, 1979.

[11] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of POPL '11*. ACM, 2011.

[12] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *Proceedings of POPL '11*. ACM, 2011.

[13] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Procedings of FoVeOOS '10*, LNCS. Springer, 2010.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of CCS '11*. ACM, 2011.

[15] P. Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *Proceedings of VMCAI '14*, LNCS. Springer, 2014.

[16] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Proceedings of TACAS '04*, LNCS. Springer, 2004.

[17] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proceedings of PLDI '08*. ACM, 2008.

[18] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of PASTE '05*. ACM, 2005.

[19] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings CAV '09*, LNCS. Springer, 2009.

[20] M. Marron, R. Majumdar, D. Stefanovic, and D. Kapur. Shape analysis with reference set relations. In *Proceedings of VMCAI '10*, LNCS. Springer, 2010.

[21] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of ESOP '05*, LNCS. Springer, 2005.

[22] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[23] T. Nguyen, S. Rumee, C. Csallner, and N. Tillmann. An experiment in developing small mobile phone applications comparing on-phone to off-phone development. In *Proceedings of USER '12*, 2012.

[24] H. Oh, L. Brutschy, and K. Yi. Access analysis-based tight localization of abstract memories. In *Proceedings of VMCAI '11*, LNCS. Springer, 2011.

[25] É. Payet and F. Spoto. Static analysis of Android programs. *Information and Software Technology*, 54(11):1192 – 1201, 2012.

[26] X. Rival. Understanding the origin of alarms in Astrée. In *Proceedings of SAS '05*, LNCS. Springer, 2005.

[27] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[28] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming*, LNCS. Springer, 2013.

[29] N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of SPLASH/Onward! '11*. ACM, 2011.

[30] D. Wolber, H. Abelson, E. Spertus, and L. Looney. *App Inventor*. O'Reilly Media, 2011.

[31] X. Xiao, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal. User-aware privacy control via extended static-information-flow analysis. In *Proceedings of ASE '12*. ACM, 2012.